MASTER EN SCIENCES INFORMATIQUES

Contribution au projet PRISM : amélioration du whiteboard, un outil multimédia de haut niveau

Van Asten, Michel

Award date:
1995

FACULTES UNIVERSITAIRES N.D. DE LA PAIX, NAMUR
INSTITUT D'INFORMATIQUE

Année académique 1994-1995

CONTRIBUTION AU PROJET PRISM : AMELIORATION DU WHITEBOARD, UN OUTIL MULTIMEDIA DE HAUT NIVEAU

Michel Van Asten

Mémoire présenté pour l'obtention du grade de Maître en Informatique

Je voudrais ici remercier l'ensemble des personnes qui m'ont aidé dans la réalisation de ce mémoire,
Monsieur le professeur J. Raemakers qui m'a proposé le sujet et a suivi sa réalisation avec attention et bienveillance,
Monsieur R. Cotet qui a toujours trouvé le temps pour répondre à mes questions,
Monsieur Toutain de l'ESNT Bretagne qui m'a accueilli à Rennes,
Oliver Huber qui m'a aidé à comprendre PRISM et le Whiteboard et m'a fait découvrir la ville de Rennes,
Mon épouse Pascale pour son soutien de tous les instants.

Résumé

Ce mémoire a pour objet la modification du Whiteboard, un outil multimédia de haut niveau conçu dans le cadre du projet PRISM.

PRISM est l'acronyme de Plate-forme distribuée pour l'intégration de services multimédia. La conception d'une application multimédia distribuée n'est pas une chose simple, surtout lorsque l'environnement matériel et logiciel est hétérogène. Les objectifs du projet PRISM sont de masquer cette complexité tout en offrant un service de qualité. La plate-forme PRISM se compose de trois blocs : (a) le bloc multimédia où l'on trouve principalement un certain nombre de services de haut niveau (son, image animée, espace de travail commun...), (b) le bloc de distribution, accessible par l'intermédiaire d'une API, et qui autorise le transfert de données multimédia et (c) le bloc de gestion qui gère l'ensemble des services multimédia proposés et demandés (conférence...).

Le Whiteboard est un des outils de haut niveau du bloc multimédia. Il offre un espace de travail commun. Il est implémenté en C dans l'environnement UNIX XWindow.

Les modifications que nous avons introduites concernent la gestion de la couleur et l'extension du nombre de formats utilisables pour l'affichage d'images préenregistrées

L'interface homme-machine de l'application a par ailleurs été modifiée pour prendre en compte les remarques des premiers utilisateurs.

En vue d'évaluer les possibilités d'implémentation du Whiteboard dans l'environnement Microsoft-Windows, nous avons mis au point un prototype de "Win Whiteboard". Le prototype est opérationnel et autorise la réception de "messages graphiques" par l'intermédiaire du protocole UDP. L'accès au réseau est réalisé par l'intermédiaire de l'interface Windows Socket.

Abstract

The object of this work is the modification of "Whiteboard", a high-level software tool conceived in the framework of the PRISM project. PRISM is the French acronym for "Plate-forme distribuée pour l'intégration de services multimédia". The conception of a distributed multimedia software is not trivial, especially in the case of a heterogenous hardware and software environment.

The PRISM platform aims to hide such complexity at the same time offering high level services to designers. It has three components : (a) the multimedia block where we find chiefly high level tools (sound, animated pictures, common work space...), (b) the distribution block allows the transfer of multimedia data and (c) the management block which manages all requested and available multimedia services.

"Whiteboard" is one of the high level tools of the multimedia block. It offers a common work space. It is implemented in C in the UNIX X Window environment.

The modifications we have introduced concerns colour management and the extension of the number of graphical format for the display of saved images.

We have also modified the human-machine interface of the software to take into account the remarks made by its first users.

To evaluate the possibility of implementing a version of "Whiteboard" in the Microsoft Windows environment, we have developed a prototype. This prototype is now operational and allows the receiving of "graphical messages" through the UDP protocol. Access to the network is realized by using the Windows Sockets interface.

TABLE DES MATIERES

1. INTRODUCTION

Dans l'environnement informatique actuel, on trouve une variété énorme de réseaux, de logiciels et de matériels. Pour des données classiques, la gestion de cette hétérogénéité n'est déjà pas une chose simple, elle l'est d'autant moins pour des applications qui font appel simultanément à plusieurs types de données tels que le son, l'image statique et l'image animée.

C'est dans le but de répondre à ce défi qu'a été lancé le projet PRISM ("Plate-forme répartie pour l'intégration de services multimédia") par l'ENSTB (Ecole Nationale Supérieure de Télécommunication de Bretagne). L'objectif de ce projet est d'offrir une solution tant pour l'utilisateur final que pour le concepteur d'applications multimédia.

La conception d'outils multimédia de haut niveau est un des objectifs du projet. Un de ces outils est le Whiteboard. Cet outil permet, dans sa version actuelle (O. HUBER,5), à un professeur ou un conférencier de montrer des images préenregistrées ainsi que des dessins tracés au moment de la présentation à un ensemble de postes interconnectés.

Notre contribution au projet est la suivante : Au stade initial de notre travail, une première version de l'application existait dans l'environnement UNIX X-Window mais elle présentait plusieurs limitations : elle ne pouvait être lancée sur des postes équipés d'écrans noir et blanc et un seul format de fichier graphique était reconnu.

La première partie de ce travail a été d'implémenter un algorithme de dégradation d'image et d'offrir un mécanisme simple de conversion de format d'image. Par ailleurs, ne disposant pas des librairies X-Window sur le système SUN OS pour lequel l'application initiale était conçue, l'application a été portée sur ULTRIX 3.4.

Un essai d'utilisation du Whiteboard dans le cadre d'une application de téléséminaire existe. L'interface de cette application a donné lieu à des critiques. Une seconde partie du travail a donc été de tenir compte de ces remarques dans l'application existante. Une bonne partie des solutions proposées sont maintenant implémentées.

L'objectif du projet PRISM étant le développement d'applications multimédia dans un environnement hétérogène, il nous a semblé intéressant d'examiner quelques aspects de l'adaptation du Whiteboard à l'environnement Microsoft Windows. Deux aspects ont été étudiés plus particulièrement : la conception de l'API de distribution et la conception de l'interface. Notre objectif était de mettre en évidence les particularités du système et de montrer comment cette API pouvait les masquer. Au point de vue de l'interface homme-machine, une démarche analogue a été suivie. Cette troisième partie comporte l'implémentation dans

l'environnement Windows d'un ensemble de fonctions de l'API de distribution de PRISM ainsi que l'implémentation d'un prototype de Whiteboard (Win Whiteboard).

Nous présenterons dans un premier temps les caractéristiques de la plate-forme PRISM. Dans un second temps, quelques exemples d'applications distribuées seront comparés afin d'en dégager les similitudes et les oppositions. Le Whiteboard sera ensuite étudié en insistant successivement sur son interface, la gestion de la couleur et des images pour terminer par les communications. Les modifications apportées à ces trois points seront alors introduites. Nous discuterons enfin des possibilités d'implémentation du Whiteboard dans l'environnement Microsoft Windows.

2. LA PLATE-FORME PRISM

2.1 DEFINITION

PRISM est l'acronyme de "Plate-forme répartie pour l'intégration de services multimédia". Le but de ce projet est de fournir des facilités pour le développement d'applications multimédia dans un environnement hétérogène.

La plate-forme PRISM est utilisée pour introduire des applications multimédia sur des stations de travail existantes sans nécessiter l'acquisition d'un matériel particulier et coûteux sur chacune. Elle offre des services réseau adaptés à la nature particulière des objets multimédia (multicast, datagramme sécurisé...) et des outils de haut niveau pour manipuler et éditer ces objets. Elle peut être utilisée pour définir des applications distribuées sur LAN et WAN.

Un autre objectif du projet PRISM est de tester et de mesurer les délais de transmission et les taux d'erreur introduits par les protocoles du réseau.

Dans un premier temps, nous présenterons l'architecture de la plate-forme PRISM segmentée en trois "blocs" dont nous détaillerons les fonctionnalités. Nous détaillerons ensuite les caractéristiques du protocole PTP.

2.2 SEGMENTATION DE L'ARCHITECTURE PRISM

PRISM a été développé, à l'origine, à partir de la notion de domaine multicast. Un domaine multicast est composé de toutes les stations que l'on peut potentiellement atteindre avec un seul paquet et est généralement composé de LAN interconnectés par des "bridges".

PRISM gère les applications multimédia dans un seul domaine. Plusieurs plates-formes PRISM peuvent être interconnectées en utilisant des réseaux synchrones (N-ISDN, ATM...) ou à commutation par paquet (Internet, X25,...).

L'architecture PRISM a, par ailleurs, été définie de façon à permettre la gestion de l'hétérogénéité et à pouvoir s'adapter à l'évolution des standards multimédia. Les trois blocs constitutifs de la plate-forme PRISM sont repris sur la figure 2-1.

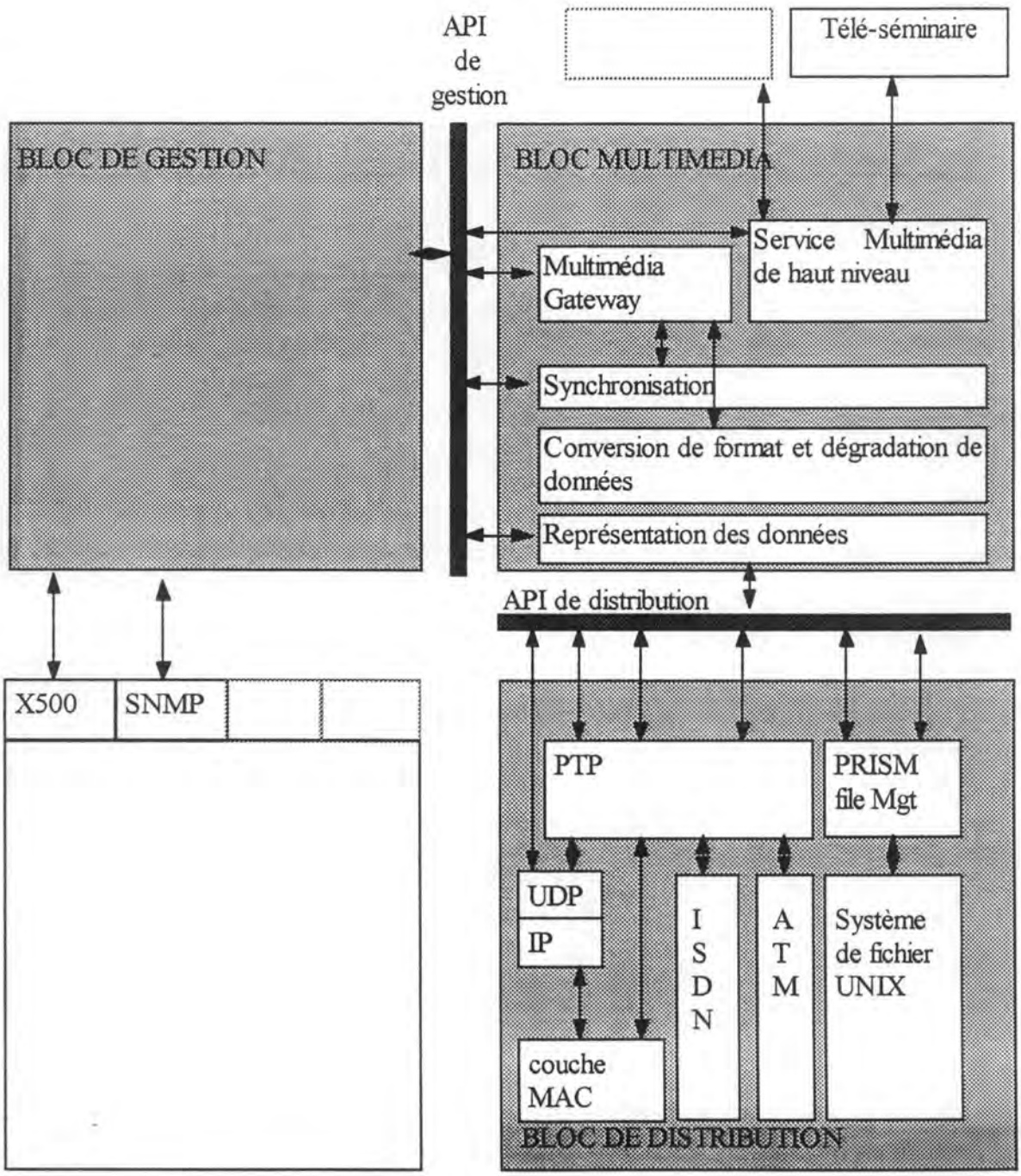


Figure 2–1 : Architecture de la plate-forme PRISM

- Le bloc multimédia gère l'information multimédia en introduisant la synchronisation, la conversion de format et la dégradation de données.
- Le bloc de gestion détermine le routage, la collaboration entre applications et est utilisé pour négocier la Qos (Quality of service). Il utilise des facilités réseau telles que les "directory X.500" ou les services de gestion de réseau SNMP.
- Le bloc de distribution définit la façon de transmettre les données sur le réseau ou de stocker les données en respectant les caractéristiques du multimédia. Il n'impose pas une manière unique pour transmettre et stocker les données.

Deux API sont définies pour établir les liens entre ces trois blocs :

- L'API de distribution entre le bloc multimédia et le bloc de communication.
- L'API de gestion entre le bloc multimédia et le bloc de gestion.

2.2.1 LE BLOC DE DISTRIBUTION DE PRISM

Gérer des données multimédia va obliger les applications à consommer les données à la vitesse où elles sont produites. Ceci implique des changements dans les caractéristiques des réseaux à commutation par paquets et dans les protocoles. Ces changements sont liés à la perception humaine des informations : le cerveau humain est en effet plus sensible aux contraintes de temps qu'aux erreurs dans les données. Malgré la perte d'un paquet dans la transmission du son, un message restera compréhensible mais si le protocole essaie de retransmettre le paquet, il introduira une désynchronisation et le son deviendra non interprétable.

De façon à ne pas limiter la plate-forme à certains environnements, plusieurs « espaces » ont été définis, un espace étant un moyen de transférer de l'information d'un point à un autre. Dans le projet, les espaces suivants existent :

- PRISM_INTERNET: le protocole utilisé est UDP/IP.
- PRISM_FILE: les données sont stockées en utilisant le système de fichier, ce qui permet, par exemple, de réutiliser une conférence préalablement enregistrée.
- PRISM_PRISM: un protocole adapté au multimédia est utilisé, il est implémenté juste au-dessus de la couche MAC.

D'autres espaces non actuellement définis pourraient être:

- PRISM_IVS: le protocole de transmission IVS est implémenté au-dessus de UDP/IP.
- PRISM_ISDN: le protocole utilise un lien ISDN.
- PRISM_ATM: des exigences de grands débits seront pris en compte.

Remarquons que dans la figure 2-1 le protocole PTP est implémenté au-dessus de ATM et d'ISDN, les services offerts par ces deux protocoles n'étant pas jugés suffisants pour le transfert de données multimédia.

L'espace ainsi défini est divisé en canaux. Un canal est une adresse spécifique dans un espace qui contient des informations multimédia. Un canal peut être :

- une adresse Internet de classe A, B, C ou D (PRISM_INTERNET),
- un nom de fichier (PRISM_FILE),
- un numéro de téléphone (PRISM_ISDN).

Un flot est une partie de canal qui contient des informations monomédia, celles-ci peuvent être multiplexées. Un flot est en mode simplex (unidirectionnel). Il est identifié par un numéro, un canal, et pour des raisons de sécurité par une adresse d'origine. Actuellement, ces numéros sont assignés de façon statique mais dans des versions ultérieures, de façon à gérer l'évolution des standards et l'hétérogénéité, cette assignation sera dynamique.

2.2.2 PRISM TRANSMISSION PROTOCOL (PTP)

2.2.2.1 Caractéristiques générales

Ce protocole est destiné au LAN. Pour d'autres types de réseau, d'autres protocoles doivent être utilisés. On suppose que :

- Le réseau offre des facilités multicast (les transmissions sont de type 1->n) et l'utilisateur ignore quelles stations recoivent les données.
- Le réseau est fiable et l'émetteur recevra des acquittements négatifs uniquement si un paquet est perdu.
- L'accès au réseau est déterministe.
- Le protocole réseau peut gérer les priorités.

Le protocole est implémenté au-dessus de la couche MAC et est utilisé pour transférer des informations ayant des contraintes de temps.

PTP a été conçu dans le but d'exploiter la propriété de tolérance aux pertes que présentent les données multimédia. On distingue au sein du flux d'informations un trafic tolérant les pertes et un trafic sensible. Un mécanisme de fiabilisation peut être appliqué au second.

Par ailleurs, le protocole permet un découplage des débits d'arrivée et de consommation des données. Ainsi, des machines différentes peuvent recevoir des versions plus ou moins dégradées des informations au sein d'un même domaine PRISM sans que la qualité de réception des autres machines ne soit affectée.

2.2.2.2 Le protocole

PTP est structuré en trois niveaux qui sont chargés de diffuser l'information, de fiabiliser la diffusion et finalement de stocker et délivrer les données aux applications réceptrices.

a. DIFFUSION

Le choix de pratiquer la diffusion (multicast) est dicté par le type des applications multimédia supportées qui impliquent plus de deux utilisateurs. Par ailleurs, lorsque cette diffusion est réalisée au niveau un ou deux, l'économie de ressources réseau n'est pas négligeable.

b. FIABILITE

Les données multimédia sont par nature résistantes aux pertes, suite à la redondance qui les caractérise mais également suite à la nature du récepteur final (l'être humain). La résistance aux pertes est toutefois maximale pour les données brutes (sans compression).

Plusieurs techniques sont utilisables pour fiabiliser les communications : acquittement/retransmission ou techniques préventives (introduction d'une redondance contrôlée des données). Le choix posé ici est d'utiliser des acquittements négatifs et de retransmettre les paquets perdus. Le contrôle des pertes est réalisé par numérotation et est effectué de façon particulière par PTP. Il est en effet possible d'inclure dans chaque paquet du trafic non fiable la numérotation des paquets devant être absolument reçus. Lorsqu'une perte est détectée, chaque récepteur du flot émet un acquittement négatif. L'entité émettrice traite le premier acquittement reçu en envoyant à nouveau les paquets puis ignore les autres requêtes relatives aux mêmes pertes pendant un certain délai.

c. GESTION DE LA MEMOIRE

Les protocoles traditionnels sont destinés à des communications point-à-point sans contrainte temporelle ; une seule entité réceptrice peut consommer les données stockées par le protocole et elle les consomme à son rythme propre. La gestion en mémoire de ces données est organisée selon le principe d'une file d'attente avec un contrôle de débordement impliquant, lorsque la file est saturée, la perte des données entrantes. Cette politique est inadaptée aux données multimédia car :

- Plusieurs entités de niveau application peuvent recevoir les mêmes données.
- Un flux multimédia présente de fortes contraintes temporelles et la politique traditionnelle favorise les données obsolètes.

Il faut donc un mécanisme de stockage des données entrantes qui soit susceptible de délivrer les informations à un nombre arbitraire d'applications.

Les contraintes temporelles sont respectées découplant le débit d'arrivée des données du débit de consommation : un flux multimédia peut être considéré comme une séquence d'unités d'information (UI), une UI étant un ensemble de données partageant une sémantique commune au point de vue de leur traitement par une application (exemple : une suite d'images MPEG). Pour la transmission, les UI seront morcelées en plusieurs paquets. Chaque paquet contient ainsi le numéro de l'UI et la valeur du décalage dans celle-ci. Ce principe est illustré par la figure 2.2.

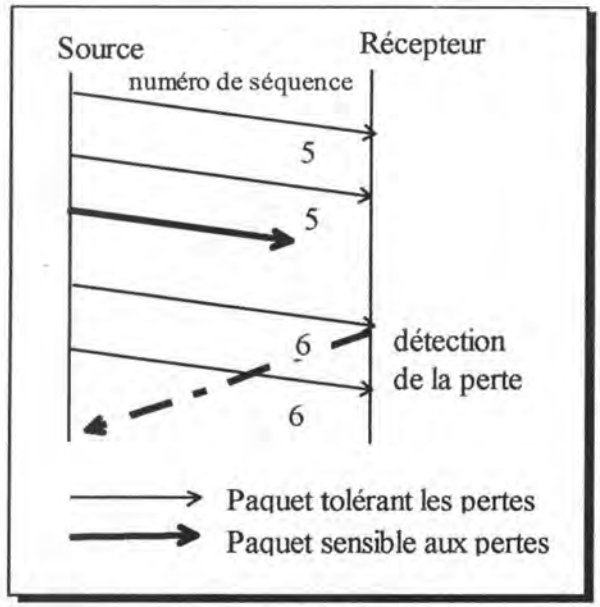


Figure 2–2 : Détection des pertes

La structure utilisée pour le stockage des UI est un tampon circulaire (liste chaînée refermée sur elle-même où chaque élément de mémoire est susceptible de stocker une UI). Un pointeur d'insertion parcourt cette liste et désigne l'emplacement de l'UI entrante. Cette UI écrase éventuellement l'UI la plus ancienne.

La politique de gestion des tampons circulaires est la suivante : l'élément de mémoire courant est rempli avec l'UI en cours de réception. Lorsque la perte d'un paquet non fiabilisé se produit, la perte est ignorée, c'est l'application destinataire qui résoud le problème. Si c'est un paquet fiabilisé qui est perdu, il faut attendre sa retransmission pour délivrer l'UI à l'application.

Il est important de remarquer que puisque la retransmission d'un paquet fiabilisé demande un certain délai, la taille du tampon circulaire va déterminer la probabilité de recevoir le paquet réémis avant que le déplacement du pointeur d'insertion n'induise sa destruction par manque de place. Par ailleurs, la taille de ce même tampon détermine le délai entre la réception des données et leur présentation. Il existe donc un compromis entre la qualité globale des données et le délai de livraison aux applications. La spécification de cette qualité de service est du ressort des applications.

Le protocole donne aux applications le moyen de faire varier dynamiquement la taille des tampons et ceci sans interférence entre elles, même pour des applications situées sur une même machine.

Pour offrir ce service, la solution qui a été retenue est d'opérer un décalage du pointeur d'insertion (figure 2-3) dans le sens du chaînage pour obtenir des UI plus récentes. Les UI plus récentes étant également de moins bonne qualité, il faut permettre à chaque application de préciser son propre décalage.

2.2.3 LE BLOC MULTIMEDIA

Au sommet des services offerts par ce bloc se trouvent les applications multimédia vues par l'utilisateur. Une application multimédia est composée de services de haut niveau capables de traiter un média (image animée, son...). Les services actuellement définis sont:

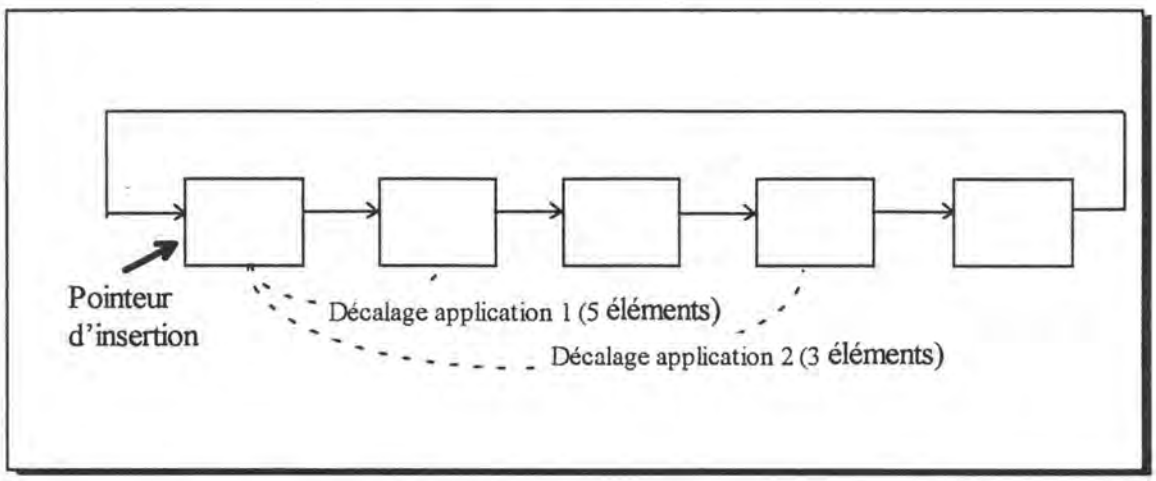


Figure 2–3 : Tampon virtuel par décalage

- le son codé en "μ-law",
- un espace commun (Whiteboard),
- l'image animée codée en H.261.

Ces services peuvent être groupés pour former de nouvelles applications.

Par exemple, un service de télé-enseignement est composé :

1. d'une application enregistrant le son d'un côté et le reproduisant de l'autre,
2. d'un espace de travail commun pour montrer une image à tous les élèves.

Pour travailler ensemble dans un espace commun, un service de contrôle des participants devra être ajouté.

Les outils de haut niveau du bloc multimédia accèdent à l'API de distribution par l'intermédiaire du bloc de gestion, comme celui-ci est en cours de développement, elle accède en réalité actuellement directement à l'API de distribution.

Au même niveau se trouvent les MMG (MultiMedia Gateway). Les MMG permettent à un service de haut niveau d'accéder à une information qui lui est inaccessible. Cette information peut être inaccessible car le format des données transmises à un moment est incompatible avec

le matériel disponible localement. Remarquons que l'accès au MMG ne se fait pas directement, il se fait par l'intermédiaire du bloc de gestion qui lui dispose des informations concernant l'ensemble des services offerts.

Les MMG réalisent les opérations de conversion entre formats par l'intermédiaire de la couche de conversion.

La couche de représentation a connaissance de la nature des données véhiculées. Les données sont mises à ce stade sous la forme de paquets qui sont transmis par le bloc de distribution. La taille des paquets peut être gérée par le bloc de gestion.

2.2.4 LE BLOC DE GESTION DE PRISM

Ce bloc est en cours de développement, ses caractéristiques sont donc susceptibles d'évoluer sensiblement.

Il connait toutes les applications PRISM présentes dans un domaine multicast et communique avec elles. Il interagit avec le bloc multimédia de quatre façons différentes :

- Avec les applications PRISM du bloc multimédia : Les applications enregistrent les services multimédia dans le bloc de gestion et décrivent la qualité minimale de service ainsi que l'importance relative de chaque flot. Dans une application comportant du son et de l'image animée, le son est généralement plus important que l'image, si un étranglement se produit, le bloc de gestion peut décider de laisser tomber certains paquets de l'image.
- Avec la couche de routage : Le bloc a accès aux informations à travers les services X.500. Cette couche peut déterminer quel hôte distant est le meilleur pour obtenir les données multimédia.
- Avec la couche de conversion : Elle organise la dégradation appropriée des données. Si deux applications du même domaine multicast désirent partager leur largeur de bande, elles doivent coopérer pour adapter leur qualité de service. Avec SNMP, le bloc de gestion est informé du taux de perte des paquets et peut adapter la qualité de service.
- Avec la couche de représentation : Le bloc optimise la taille des paquets de données, plus petits sont les paquets, plus les temps de transmission sont bons, mais de petits paquets surchargent le réseau.

Remarquons que l'architecture de ce bloc n'est pas encore arrêtée et que son implémentation actuelle est très limitée. Le bloc de gestion tel qu'il existe actuellement se contente de reprendre l'ensemble des fonctions définies dans le bloc de distribution. En conséquence, le niveau "service multimédia" représenté par exemple par le Whiteboard fait directement appel aux fonctions de l'API de distribution.

2.2.5 L'INTEGRATION D'APPLICATIONS DANS L'ENVIRONNEMENT PRISM

L'application principale existant actuellement est le « téléséminaire ». Cette application est utilisée pour distribuer, stocker et rejouer des cours et des conférences. Le conférencier est dans un studio, un show-room ou devant sa station de travail personnelle. Les auditeurs sont également devant leur propre station de travail ou dans une salle de classe. Le conférencier utilise des documents digitalisés (slides) dont les auditeurs doivent obtenir une copie avant le début de la leçon. Les actions du conférencier (mouvements de la souris, dessin...) sont envoyées en temps réel aux auditeurs. Cette application a été choisie car elle n'engendre pas un volume de données très élevé sur le réseau et la synchronisation entre les différents objets multimédia n'est pas significativement importante.

3. APPLICATIONS DISTRIBUEES ET TRAVAIL COOPERATIF : QUELQUES EXEMPLES D'ARCHITECTURE

3.1 INTRODUCTION

L'objectif de ce chapitre est de présenter des exemples d'architecture logicielle pour des applications distribuées ainsi que la justification des choix opérés.

Parmi les applications distribuées, de gros efforts de recherche sont entrepris depuis quelques années dans le domaine du CSCW (computer-supported cooperative work). C'est donc tout naturellement sur ce sujet que l'on trouve beaucoup d'exemples d'architecture d'applications distribuées. Les applications qui pourront être développées à partir de la plate-forme PRISM devraient entrer dans cette catégorie.

Ce chapitre est basé sur l'analyse de trois travaux récents ayant pour objet la mise en oeuvre d'outils permettant le travail coopératif. Le terme outil est volontairement imprécis car selon les cas, il s'agit d'applications complètes et/ou d'outils destinés à aider au développement d'applications.

3.2 LE PROJET CONUS [1]

3.2.1 PRINCIPES GENERAUX

Deux grandes options sont possibles en matière de conception d'application de groupe :

1. La première est d'utiliser des applications mono-utilisateur classiques (traitement de texte, tableur...) dans lesquelles un groupe d'utilisateurs peut travailler en commun par l'intermédiaire de « télépointeurs ». Ce type d'approche permet aux utilisateurs de travailler de façon tout à fait classique, ce qui est un avantage, mais tous les utilisateurs devront travailler sur le même document et sur la même page. La collaboration entre utilisateurs n'est donc pas directement soutenue par l'application, elle est implicite. La figure 3-1 montre un exemple d'architecture correspondant à cette approche.
2. La deuxième approche consiste à créer des applications spécifiques au travail de groupe. Dans ce cas, la collaboration est explicite. Les GDSS (Group Decision Support System) entrent dans cette catégorie.

---

[1] M. WEBER, J. SCHEITZER, G. VÖLKSEN, 12.

Examinons maintenant plus en détail la première option et les fonctionnalités des différents modules définis dans la figure 3-1. Cette architecture définit un cadre pour le développement du projet CONUS.

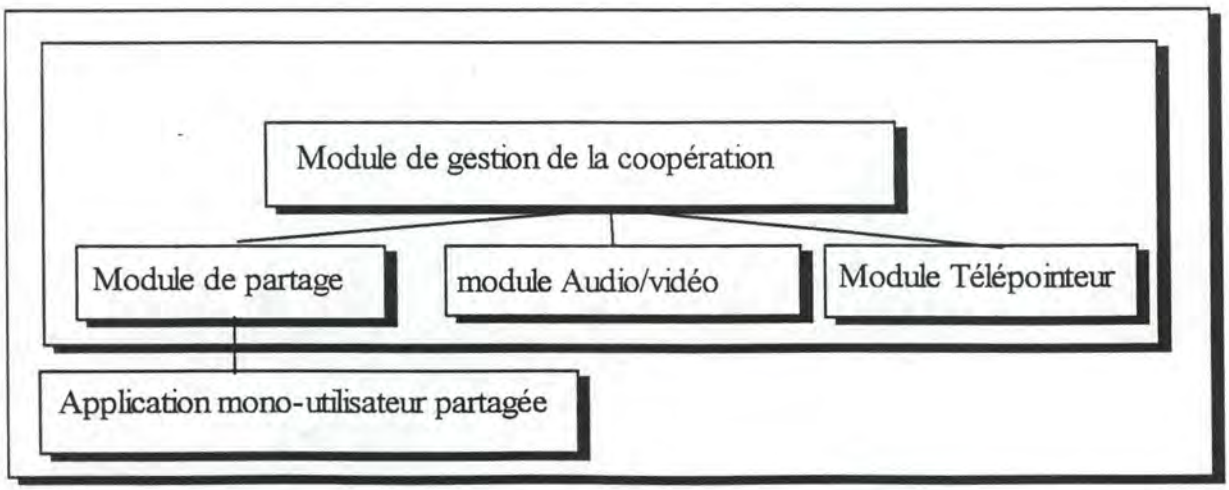


Figure 3–1 : Exemple d'architecture à collaboration implicite

Le module de gestion de la coopération (CMM, Cooperation management module)

Ce module a pour fonction d'administrer un ensemble dynamique de participants qui peuvent entrer et sortir durant la session. La personne qui initialise les sessions définit les politiques de session et invite les collaborateurs potentiels à participer. Il y a trois genres de session :

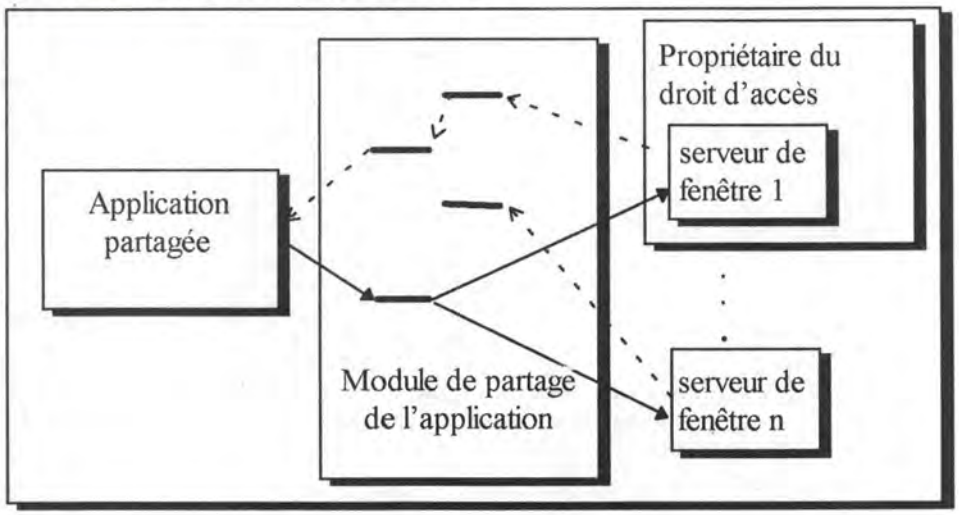
(1) ouverte, aucune permission n'est requise pour y participer,

(2) joignable, une autorisation doit être obtenue pour y participer,

(3) fermée, seuls ceux qui sont invités peuvent participer,

Le module de partage de l'application

A : Distribution des interfaces utilisateur



B : Distribution et répliaction de l'application

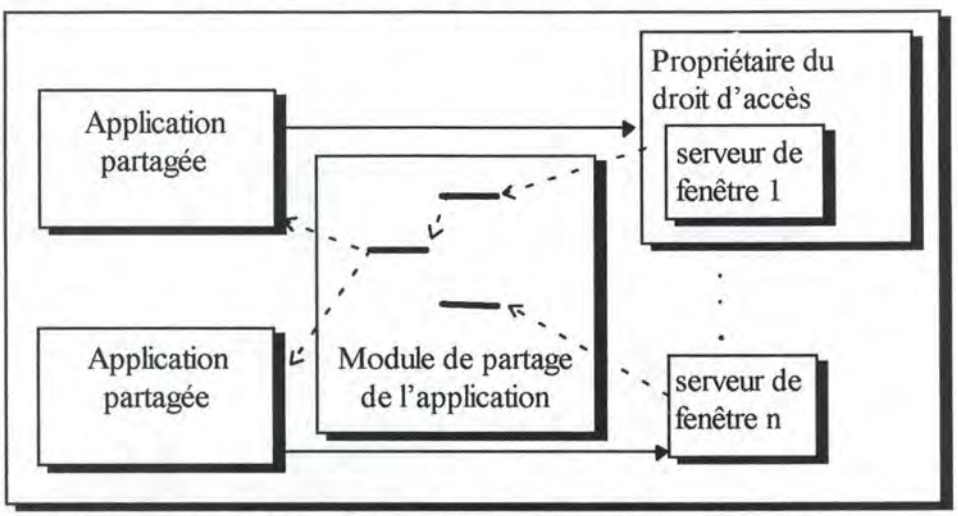


Figure 3–2 : Deux formes de partage de l'application

Ce module diffuse les sorties vers l'ensemble des participants tout en limitant le droit à l'entrée d'informations à un seul. L'idée est d'intercepter le flux lié au protocole de gestion des fenêtres entre l'application et le serveur. Deux variantes majeures d'un Application Sharing Module (ASM) sont possibles (voir figure 3-2) :

- L'application est exécutée sur une machine et son interface utilisateur est présente sur la machine de chaque utilisateur.
- Des copies de l'application sont exécutées sur toutes les machines en même temps. Les accès (autorisés) à l'application sont multiplexés et envoyés de façon synchrone aux autres instances de l'application distribuée.

Dans les deux cas, le contrôle de l'accès est implémenté dans l'ASM. Mais les différences sont notables :

- Dans le premier cas, ni l'application ni les données ne sont dupliquées, ce qui nécessite des transferts importants sur le réseau. L'ASM doit, par ailleurs, tenir compte de l'hétérogénéité du matériel pour adapter le flux d'information du protocole aux caractéristiques du site de destination.
- Dans le second cas, la duplication limite les transferts. Les actions de l'utilisateur sont directement dirigées vers l'application locale. Cependant, toutes les données de l'environnement de travail doivent être identiques sur chaque machine. De plus, certaines applications ne sont pas nécessairement accessibles sur toutes les machines.

Le module de « télépointage » (TPM Telepointing module)

Quand un télépointeur est déplacé dans une fenêtre appartenant à une application partagée, le TPM suit les mouvements du télépointeur utilisé localement et les diffuse sur les autres sites.

Des transformations de coordonnées doivent être réalisées pour que le télépointeur apparaisse à la même place dans la fenêtre partagée et sur le site source. De façon à limiter la charge du réseau, une fréquence de mise à jour ajustable est prévue pour le télépointeur. Les télépointeurs peuvent également être synchronisés avec les flux de son provenant du module audio/vidéo (AVM) conférence. Ceci peut être réalisé en retardant le moment de la présentation du télépointeur jusqu'à ce que le son correspondant soit reçu ou en laissant de côté des positions intermédiaires du télépointeur.

Le module Audio/vidéo conférence (AVM)

Le travail coopératif requiert le support de moyens audiovisuels. Le module AVM est basé sur un service de communication audio/vidéo qui réalise le multiplexage et la diffusion des sources audio et vidéo vers l'ensemble des participants. Il inclut des possibilités de traitement du son et de l'image telles que la compression et la décompression.

Architecture centralisée ou décentralisée

Après avoir défini les différents modules de l'architecture, il s'agit de déterminer leur localisation.

- Approche centralisée : Dans ce cas, l'implémentation suit le modèle client-serveur et chaque module précédemment décrit est en réalité un processus serveur qui s'exécute sur une machine spécialement affectée à cette tâche. De cette manière, même les tâches qui

pourraient être exécutées localement engendrent une requête au serveur. L'avantage de cette approche centralisée est que l'état du système reste toujours cohérent. Le désavantage est la charge importante du réseau.

- Approche distribuée : Une instance (agent) de chaque module est exécutée sur chaque machine et chaque instance a les mêmes potentialités. La spécificité (matérielle et logicielle) de chaque machine est masquée par l'intermédiaire de protocoles spécifiques. L'inconvénient majeur de cette solution est le risque d'incohérence.

3.2.2 LE SOUS-SYSTEME CONUS (COOPERATIVE NETWORKING FOR GROUPS)

Si nous reprenons les différentes caractéristiques des CSCW mises précédemment en évidence, un système tel que nous venons de le décrire présente des lacunes (voir figure 3-1). L'architecture proposée peut néanmoins servir de cadre pour une implémentation particulière. Le sous-système CONUS intègre les différents modules décrits et est conçu de façon à être facilement étendu. Il intègre la gestion d'applications où la collaboration est explicite.

Interfaces pour " l'interaction homme-homme"

L'interface pour les systèmes CSCW inclut des techniques pour l'interaction homme-machine et l'interaction entre hommes via la machine. Si le premier type d'interaction est simple à comprendre, le second nécessite quelques explications. Le projet inclut l'implémentation d'un espace de travail commun où les interactions ont lieu. Les données accessibles dans cet espace sont publiques, à l'extérieur de cet espace, les applications et les données sont privées.

L'espace de travail commun est visible et identique pour tous les utilisateurs. Il est clairement délimité sur l'écran de chaque utilisateur. Pour mettre une application ou un fichier en commun, il suffit de déplacer son icône dans l'espace commun.

Chaque utilisateur peut travailler de façon autonome sur un document précédemment partagé. Pour ce faire, il transfère une copie de ses données dans l'espace de travail commun. Ce travail autonome peut engendrer des incohérences lors d'une connexion ultérieure. Le système vérifie alors les incohérences éventuelles et les notifie à l'ensemble des utilisateurs. La résolution des conflits est laissée actuellement aux individus.

Applications partageables collaboratives

Dans un premier temps, les auteurs du projet se sont concentrés sur les applications au domaine de la gestion. Deux applications fondamentales ont été étudiées : un calendrier et un outil de gestion des personnes et des adresses. Ces applications sont liées entre elles. Par

exemple, l'outil "calendrier" montre une réunion; en sélectionnant l'objet réunion, les noms des participants sont affichés. En sélectionnant une personne, les données dont l'accès est public associées à celle-ci sont alors disponibles. A chaque application est associé un type particulier de fichier. Les applications communiquent entre elles pour obtenir les données complémentaires.

Base de données

Les données partagées sont organisées sous la forme d'une base de données. La personne est considérée comme l'élément central d'un système de type CSCW. Un schéma entité-association peut être construit autour de cette entité (voir figure 3-3). Toutes les associations définissent l'ensemble des activités de la personne au sein de l'organisation.

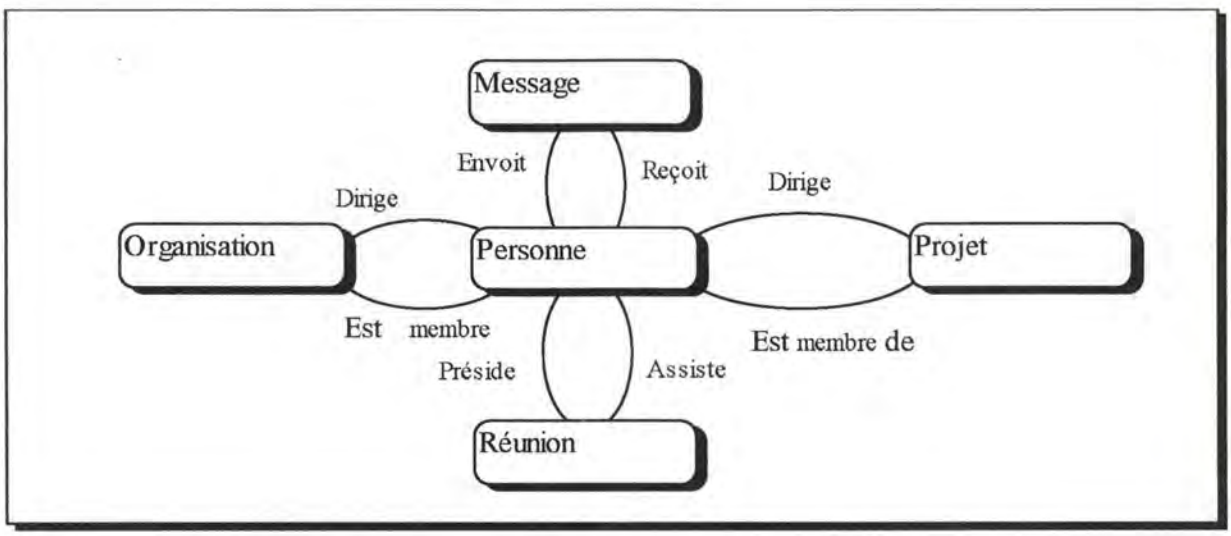


Figure 3–3 : Schéma entité-association construit autour d'une personne

Afin de permettre un travail autonome, une réplique de la base de données est présente sur le serveur et sur les machines clients. La figure 3-4 montre les différents modules du sous-système ainsi défini.

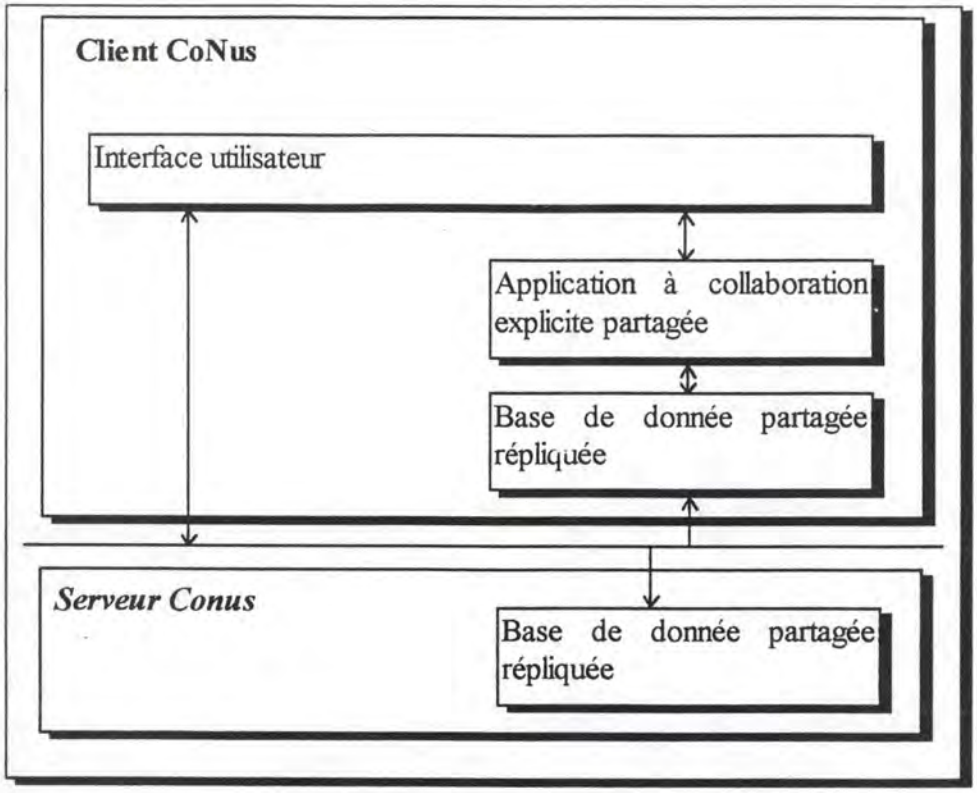


Figure 3–4 : Le sous-système CONUS

3.3 LE PROJET MEAD[2] (MULTIPLE ELECTRONIC ACTIVE DISPLAY)

3.3.1 PRINCIPES GENERAUX

L'architecture présentée ici fait partie d'un projet destiné à rechercher les applications du travail coopératif pour des contrôleurs aériens. Une des contraintes de cette architecture est d'autoriser l'expérimentation par une équipe multidisciplinaire d'un grand nombre d'alternatives quant à la conception de l'interface utilisateur.

La conception de l'interface doit être adaptée au travail de groupe. Dans le cas où elle est explicitement adaptée à cet objectif, on peut introduire la notion d'interface multi-utilisateurs. L'objectif d'une telle interface est de maintenir un contexte commun de façon à ce que l'activité d'un utilisateur se répercute sur l'activité des autres.

La manière dont l'interface multi-utilisateurs supporte le partage définit ce que l'on nomme « interface coupling ». Trois niveaux de partage de l'information peuvent être définis :

- Partage au niveau présentation : Le partage est total, tous les utilisateurs ont la même vision de données appartenant à un espace de travail commun.

---

[2] R. BENTLEY, T. RODDEN, P. SAWYER, I. SOMMERVILLE, 1.

- Partage au niveau de la vue : Le partage se fait au niveau de données mais leur présentation peut varier.
- Partage au niveau des informations : Chaque utilisateur manipule ses propres données avec une présentation strictement personnelle. Ce dernier niveau correspond à une absence de partage, il est toutefois logique de le présenter ici puisqu'il peut correspondre à un "état" temporaire d'une application coopérative.

Le projet MEAD offre une solution qui supporte ces trois niveaux de partage.

3.3.2 L'ARCHITECTURE

L'objectif est de développer une architecture qui permette la création d'interfaces multi-utilisateurs et qui, parallèlement, offre des facilités pour modifier cette interface. Cette architecture est basée sur des agents autonomes qui encapsulent les détails de la politique de partage de façon à gérer l'interface multi-utilisateurs indépendamment des aspects spécifiques à l'application. C'est une architecture hybride car d'une part, l'information partagée est conservée cohérente de façon centralisée et d'autre part, la présentation et l'interaction sont distribuées.

Les "User Display agents"

L'architecture proposée considère les informations associées aux « Display » comme des entités autonomes dont les propriétés peuvent être adaptées par les utilisateurs et les développeurs d'interface. L'état de ces entités caractérise la manière dont ces informations sont présentées. Les modifications apportées aux informations sont seules propagées. Ces entités portent le nom de "user display agent" et la part d'un écran gérée par un agent porte le nom de "user display".

Chaque agent gère un "user display" et cet UD peut être présent sur plusieurs écrans. Par ailleurs, plusieurs UD peuvent être destinés à représenter différemment la même information. L'ensemble des UD de chaque utilisateur peut ainsi être complètement différent.

Le concept d'UD permet de gérer aisément trois paramètres importants du point de vue de l'utilisateur :

- Le centrage (focus) : Les utilisateurs peuvent n'être intéressés que par un sous-ensemble de l'information, ce sous-ensemble peut évoluer.
- La représentation : Plusieurs utilisateurs peuvent utiliser les mêmes données mais avec des représentations différentes (en fonction de la tâche, de l'expérience de l'utilisateur...).
- La position : La position des représentations sur l'écran peut fournir une information importante, par exemple la position d'un avion sur un écran.

Une UD est définie par un triplet comprenant une sélection, une présentation et une composition :

- Une sélection est un ensemble d'entités d'information sélectionnées en fonction de critères de sélection. Ces critères portent sur les attributs de ces entités. Un critère de sélection pourrait porter sur les coordonnées d'un objet.
- Une présentation est un ensemble de vues pour représenter les entités de la sélection. Une vue est une représentation graphique qui définit l'apparence d'une entité d'information. Des changements dans les attributs des entités peuvent requérir des changements dans leur présentation. Par exemple, dans le cadre du contrôle aérien, on peut décider de représenter les avions privés différemment de ceux des avions de ligne.
- Une composition est un ensemble de positions représentant l'arrangement spatial de vues dans l'UD. Cette composition peut changer. Les conséquences d'un changement dans l'état d'une entité d'information ou d'un des critères sont prises en compte par l'agent du UD.

Un agent de UD (display agent) peut être membre de plus d'un ensemble de travail (ensemble des UD présentes sur un poste de travail). Chaque écran géré par cet agent subit les modifications apportées aux informations de la même manière. Pour supporter cette possibilité, une copie de cet agent peut être maintenue dans chaque ensemble de travail. Ces "surrogate" sont des agents simplifiés qui conservent uniquement l'état de la sélection, de la présentation et la composition de l'UD. La définition de l'UD (les critères) est conservée par un agent de UD maître. Ce dernier agent reçoit notification des changements subis par les entités d'information, en déduit les modifications que doivent subir les UD et les en informe.

Dans l'implémentation du projet, les entités d'information partagées sont conservées dans un "object store" et sont manipulées par un "object store server" (OSS). Ce composant conserve également les agents de UD maître. Si une nouvelle machine désire s'insérer dans le groupe, il suffit à cette dernière de contacter l'OSS pour enregistrer son existence et créer les surrogates nécessaires. Une machine peut se retirer d'un groupe de façon similaire.

Maintien de la cohérence

Les changements apportés aux données partagées peuvent affecter les composantes sélection, présentation et composition de chaque UD.

Il y a deux options pour repérer des changements dans l'état des entités d'information.

1. Les agents peuvent périodiquement vérifier l'état des entités d'information.
2. Les objets d'information peuvent notifier les changements survenus aux agents.

C'est une variante de la deuxième option qui est utilisée ici. Tous les changements subis par les objets d'information sont notifiés à un agent nommé "update handler". Cet agent

propage les modifications dans "l'object store" (voir figure 3-5) et notifie la modification aux seuls agents maîtres intéressés. Ceci implique que les agents doivent préciser au "update handler" l'ensemble d'informations qui leur est nécessaire. Ce filtrage des mises à jour par le "update handler" minimise le volume des communications.

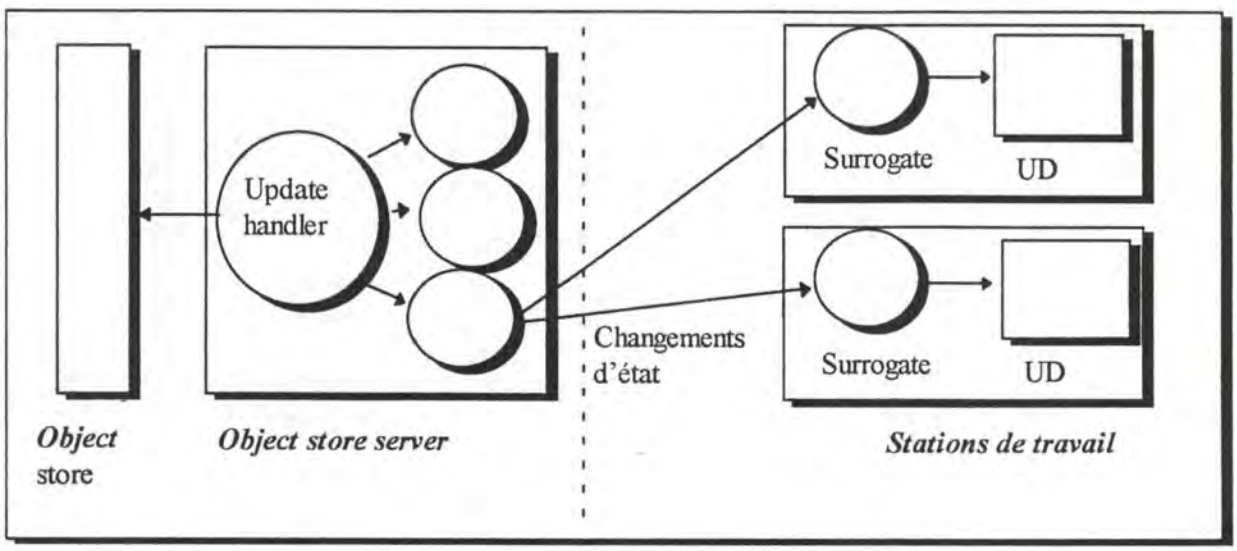


Figure 3–5 : Constituants du système MEAD

Vues multiples

L'information partagée peut être représentée dans les UD par différentes vues. Si des changements surviennent, toutes les vues doivent être mises à jour pour conserver la cohérence. Pour répondre à cet objectif, les vues sont liées aux objets qu'elles représentent. Toute modification liée à l'interaction avec l'utilisateur est ainsi répercutée sur l'information partagée. De façon à minimiser les communications entre l'OSS et les machines distantes, les objets d'information sont dupliqués et conservés localement (voir figure 3-6). Les liens sont bidirectionnels, de cette façon l'"update handler" peut répercuter à la fois les modifications liées à l'interface utilisateur et les changements externes (exemple : le déplacement d'un avion dans le cas du contrôle aérien).

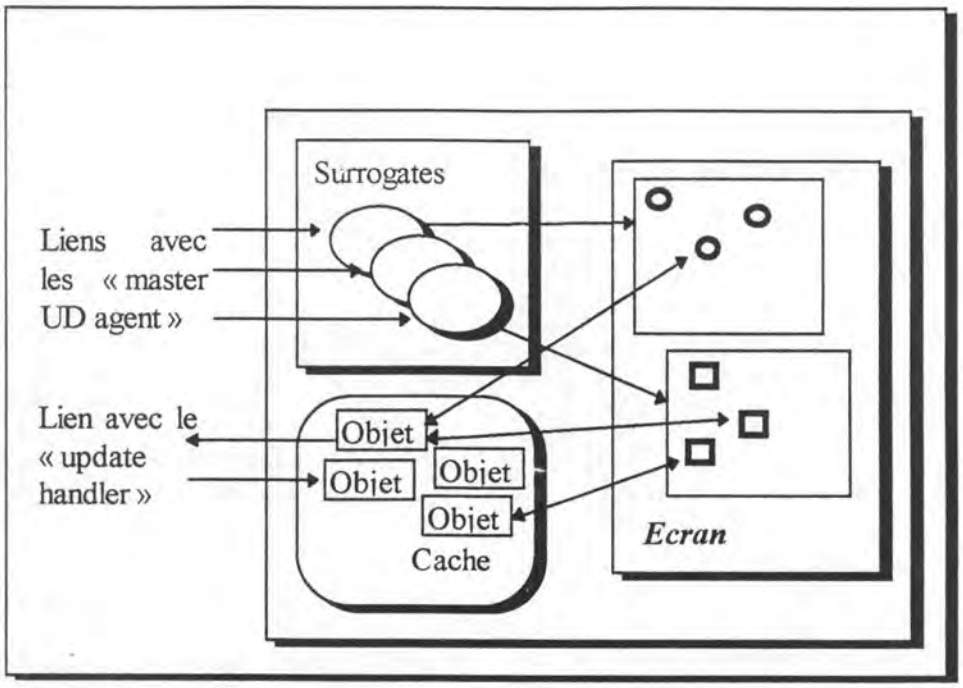


Figure 3–6 : Le mécanisme de cache local

Implémentation et perspectives

La dernière version de MEAD (version 2.0) existe sous la forme d'un prototype développé avec Objectworks/Smalltalk Realease 4.1. pour stations Sun. Il est utilisé dans plusieurs projets de recherche à l'Université de Lancaster. Un de ces projets est COMIC (Computational Mechanisms of Interaction for Cooperation) dans le cadre du programme ESPRIT. Il est utilisé pour rechercher les possibilités de partage dans les systèmes coopératifs.

3.4 LE PROJET DEEDS[3]

3.4.1 PRINCIPES GENERAUX

L'architecture client/serveur à deux étages est inadéquate dans un environnement multitâches de groupe (multitasking group environnement) car la coordination doit alors prendre en considération non seulement les interactions au sein de l'application mais également les interactions entre de multiples applications. Le design d'un tel système doit différencier les fonctions de coordination spécifique à une application particulière de celles partagées par toutes les applications.

---

[3]T. LIANG et al., 6.

L'architecture présentée ici est basée sur trois couches, elle étend l'architecture client/serveur pour inclure un "groupware server", des "application servers" et des "application clients".

La coordination

La coordination concerne des aspects multiples de l'activité de groupe, par exemple, le contrôle de processus (au sens de l'application), le contrôle de cohérence, la résolution de conflit et le contrôle de version. La complexité de la coordination est variable et dépend des éléments suivants:

- La nature des applications : Pour les auteurs, le facteur de complexité le plus important est la présence d'exigences d'interactions en temps réel car il faut introduire des contrôles complexes des accès concurrents. Les auteurs proposent par ailleurs de classer les applications de type « groupware » en trois groupes de fonctionnalité. Le tableau 3-1 synthétise les exigences de ces trois groupes.

Tableau 3-1 : Exigence des applications en terme de coordination

Type d'application	Exigences en termes de coordination
Communication	Routage, séquencement, contrôle d'accès.
Collaboration et décision de groupe	Routage, séquencement, contrôle d'accès, contrôle de version, contrôle de cohérence, contrôle de la concurrence.

- L'environnement de travail où deux aspects sont à considérer :
  - ◊ L'environnement est-il multitâche ? Dans le cas d'un environnement monotâche, les mécanismes de coordination ne concernent qu'une seule application. Les mécanismes de coordination sont plus complexes lorsque l'utilisateur peut en utiliser plusieurs simultanément.
  - ◊ L'environnement de l'utilisateur est-il modifiable ? Cela concerne des problèmes relativement simples comme un changement de couleur pour une fenêtre mais également la possibilité de travailler sur des machines d'un type différent.
- L'interaction entre agents : Le terme agent est utilisé ici dans le sens d'utilisateur du "groupware". Les formes de coordination sont multiples, elles concernent cinq types d'interaction entre agents; d'un individu à un autre individu, d'un individu à un groupe, d'un groupe à un autre groupe, d'un individu à tous (tous étant l'ensemble des utilisateurs déclarés d'un groupware) et d'un groupe à tous. Il faut également considérer le type de

relation entre les agents (relation hiérarchique ou non). Un troisième aspect est de considérer les relations entre les agents et l'application : y a-t-il une limite supérieure au nombre de participants, peuvent-ils librement accéder ou quitter l'application ...

- Le contrôle des artifacts : Les artifacts sont les objets sur lesquels les participants travaillent. Le tableau 3-2 reprend les problèmes de coordination liés aux artifacts.

Tableau 3-2 : Problèmes de coordination liés aux artifacts

Problème	Description
Gestion des privilèges	Un auteur peut-il créer, modifier ou détruire le document sur lequel il travaille ?
Gestion des artifacts	Comment définir, stocker et manipuler les objets?
Contrôle de cohérence	Comment être sûr que tous les utilisateurs reçoivent la même version du document partagé ?
Propagation de modifications	
Contrôle de concurrence	Comment mettre en place des mécanismes de blocage pour le contrôle de la concurrence?

3.4.2 DEEDS

Architecture

DEEDS est un prototype environnement multitâches. Il est basé sur une architecture en trois couches.

A. Le serveur de "groupware" : L'objectif de cette entité est de fournir les fonctions de coordination nécessaires pour toutes les applications de groupe. Les fonctions de gestion des utilisateurs sont par exemple gérées à ce niveau. Lorsqu'un utilisateur veut se joindre à une réunion électronique, le serveur de "groupware" vérifie, grâce aux profils des utilisateurs maintenus à son niveau, si ce dernier est autorisé à y participer. Il en informe ensuite le serveur de l'application correspondante.

B. Le serveur de l'application : Cette entité gère les besoins spécifiques à une application particulière. Une application de dessin de groupe nécessite un mécanisme qui divise la surface de tracé pour autoriser un travail simultané. Puisque ce mécanisme est spécifique au travail de dessin, il est implémenté à ce niveau et non pas au niveau du serveur de

"groupware". Le contrôle de l'accès aux outils de dessin est également défini à ce niveau ainsi que les divers contrôles de cohérence.

C. Le client : Son rôle principal est de fournir l'interface utilisateur et les outils de l'application.

Coordination de tâche

La coordination de tâches suit l'architecture en trois couches. Le serveur de « groupware » est responsable de l'essentiel de la coordination entre applications. La coordination intra-application est réalisée par le serveur d'application. L'essentiel de la coordination est donc réalisée au niveau des serveurs. La figure 3-7 montre les actions qui peuvent être entreprises au niveau du client dans le cas d'une application de réunion électronique ainsi que les changements d'état qui en découlent.

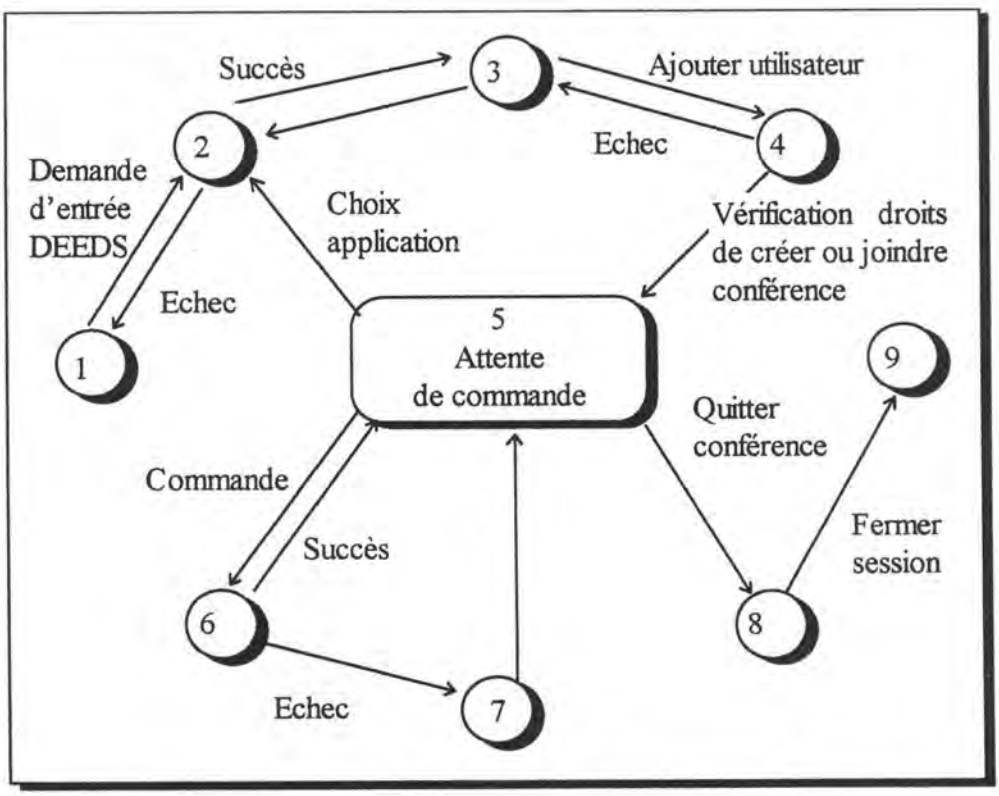


Figure 3–7 : Diagramme Etats-transitions liés à une application de réunion électronique

Le système dont il est fait mention ci-dessus est l'ensemble constitué :

- du serveur de groupware,
- du serveur de l'application de réunion électronique,
- des outils nécessaires à l'interaction avec l'utilisateur.

L'utilisateur cherche d'abord à entrer dans le système DEEDS. Cette opération est sous le contrôle du serveur de "groupware", le système passe alors dans l'état 2. L'utilisateur attend une réponse du serveur. Si l'utilisateur est valide, le système passe dans l'état 3 pour choisir une application. L'utilisateur fait une requête pour créer ou se joindre à une conférence, l'état 4 est atteint, cette opération est toujours sous le contrôle du serveur de groupware. En cas d'acceptation, l'état 5 est atteint et le système passe sous le contrôle du serveur de l'application. En quittant l'application, l'état 8 (assimilable à l'état 2) est atteint, l'utilisateur peut quitter le système (état 9) ou faire une requête pour utiliser une nouvelle application.

Implémentation

Le système comprend un noyau de « groupware » et plusieurs modules et est implémenté dans l'environnement Microsoft Windows. Borland C++ ainsi que la librairie WinSocket ont été utilisés comme outils de développement. TCP/IP est utilisé comme protocole de communication. Les applications développées à l'heure actuelle supportent le dessin de groupe, les réunions électroniques et des échanges formalisés de messages entre membres du groupe d'utilisateurs.

3.5 CONCLUSIONS

Ces trois projets partent d'une idée maîtresse spécifique, les solutions logicielles qui en découlent sont radicalement différentes :

- Projet CONUS : Le système est conçu autour d'une base de données répliquée dont le schéma est calqué sur l'activité de l'individu dans l'organisation. Le contenu de l'écran de l'utilisateur est adapté de façon à distinguer visuellement les données partagées des données privées. Au point du vue de l'implémentation, il y a réplication de l'application et de l'interface utilisateur pour limiter la charge du réseau et les problèmes de compatibilité.
- Projet MEAD : L'objectif est de séparer au maximum les problèmes de coordination et de stockage des données des problèmes de présentation des données. La solution est partiellement centralisée puisque les entités d'information sont regroupées. Mais la gestion de la présentation est gérée localement par l'intermédiaire de "surrogate".
- Projet DEEDS : L'objectif est d'isoler la gestion des utilisateurs (groupware server) de celle de l'application proprement dite (application server) et de celle de l'interface utilisateur et des outils spécifiques (application tools). La solution proposée se distingue de l'architecture client-serveur par l'ajout d'un « groupware server ». La conception d'applications spécifiques fait partie du projet.

4. LE WHITEBOARD

4.1 INTRODUCTION

Le Whiteboard est un outil de haut niveau du bloc multimédia de PRISM (O. HUBER, 5). Sa fonction est de fournir un espace de travail en commun pour un ensemble d'utilisateurs.

Le Whiteboard est en cours de développement. Actuellement, le Whiteboard permet un échange d'information d'un poste "maître" vers un ensemble de postes en attente d'information. L'outil n'offre par encore de système de gestion des utilisateurs.

Dans ce chapitre, nous présenterons successivement les caractéristiques de l'environnement de développement XWindow, l'interface de l'application, la gestion de la couleur et la communication entre applications. Les modifications que nous avons introduites seront présentées au cours de la présentation.

4.2 L'ENVIRONEMENT DE DEVELOPPEMENT XWINDOW

4.2.1 LES CONCEPTS PRINCIPAUX DE X WINDOW

4.2.1.1 Display et écran

Un display est défini comme étant une station de travail disposant de un ou plusieurs écrans et d'une souris.

4.2.1.2 Modèle Client-Serveur

X est un système fenêtré orienté réseau. Une application peut être exécutée sur une autre machine que le display. Un même display peut être utilisé par des applications s'exécutant sur plusieurs machines.

Le programme qui contrôle chaque display est appelé "serveur". Ce serveur réalise les tâches suivantes :

- Autoriser les accès au display par de multiples clients.
- Interpréter les messages qui viennent des clients.
- Transmettre les inputs de l'utilisateur au client.
- Réaliser les affichages sur le ou les écrans.
- Maintenir les structures de données complexes (curseur, polices de caractère...) qui peuvent être partagées entre clients, ce qui réduit le volume des données transférées sur le réseau.

4.2.1.3 Le protocole X

Le protocole X spécifie le contenu de chaque paquet d'information transféré du serveur vers le client et du client vers le serveur. Même lorsque le client et le serveur tournent sur la même machine, le protocole est utilisé pour la communication. Il y a quatre types de paquets utilisés par le protocole: les requêtes, les réponses, les événements et les erreurs.

- Une requête est générée par Xlib et envoyée au serveur, elle peut véhiculer des informations très variables (une couleur à modifier...). La plupart des routines de Xlib génèrent des requêtes, les seules exceptions sont les routines qui affectent uniquement les structures de données locales.
- Une réponse est envoyée lorsqu'une requête demande réponse (ce n'est pas le cas de toutes). Les routines qui demandent une réponse sont à limiter au maximum car elles diminuent les performances lorsque le réseau induit des délais importants.
- Un événement est envoyé du serveur vers le client et contient des informations à propos du déplacement de la souris ou encore d'une fenêtre à rafraîchir.
- Une erreur indique au client qu'une précédente requête est non valide. Les erreurs sont envoyées à une routine de gestion des erreurs de Xlib. Le gestionnaire par défaut affiche simplement un message. Il peut être remplacé par une routine spécifique.

4.2.1.4 Le gestionnaire de fenêtre

Le gestionnaire de fenêtre est un client X qui, par convention, possède des responsabilités particulières. Il a pour rôle de gérer les demandes en compétition pour une ressource. Généralement, il fournit une interface utilisateur pour permettre à l'utilisateur de déplacer ou de redimensionner une fenêtre. La plupart des gestionnaires de fenêtre entoure la fenêtre principale d'une application d'une barre de titre, d'un bouton de dimensionnement et d'un bouton pour l'icônifier.

4.2.1.5 Les ressources

Lors du lancement d'un client X, le gestionnaire de ressources qui fait partie de Xlib est appelé. Ce module va lire un ensemble de fichiers de configuration placés à des endroits bien définis au sein de l'arborescence d'un système de fichier UNIX. Ces fichiers sont créés en partie par le développeur et l'utilisateur. Ces fichiers permettent de définir un grand nombre d'options qui vont de la couleur d'un bouton à la fonction à associer à un raccourci clavier en passant par le choix d'une police de caractère. Chaque ressource associée à une application est définie par une paire (nom de ressource : valeur). L'ensemble des ressources d'une application est repris

sous le terme de "ressource database". Remarquons encore qu'il est possible de définir les valeurs de certaines ressources par l'intermédiaire des options de la ligne de commande.

4.2.2 PROGRAMMATION ET LIBRAIRIES DE XWINDOW

4.2.2.1 Xlib

Les applications X qui communiquent avec un serveur X le font par l'intermédaire d'appels à une librairie de routines de bas niveau écrites en langage C. Les appels à Xlib sont traduits en requêtes définies par le protocole X et transmises via divers protocoles de transport, TCP/IP est un des plus courants.

4.2.2.2 Les Widgets

La programmation sur base des primitives de Xlib est une programmation de bas niveau. Il faut explicitement prévoir la réponse à apporter aux différents types d'événements, préciser tous les paramètres d'affichage d'une fenêtre... La programmation à partir de X Toolkit simplifie notablement la programmation en apportant les concepts d'héritage et d'encapsulation, concepts associés à la programmation orientée objet :

- Grâce au mécanisme d'héritage, on ne s'intéresse qu'à la programmation des propriétés nouvelles d'un objet, il conserve toutes les propriétés de l'objet dont il hérite.
- Par encapsulation, on regroupe les structures de données et les méthodes qui les manipulent.

Les objets que manipule X Window sont appelés Widgets. Une Widget est une association entre une fenêtre et une structure de données. Cette structure de données permet d'accéder notamment aux méthodes qui agissent sur cette fenêtre. Par héritage, toutes les Widgets sont issues d'une Core Widget.

Le coeur de X Toolkit est une librairie écrite en langage C appelée Xt, parfois dénommée X Toolkit Intrinsics. L'ensemble des Widgets offert par Xt est minimal; aussi, généralement, le programmeur fait appel à l'une ou l'autre librairie construite au-dessus de Xt. OSF-Motif, Xt-Athena et OpenLook en sont des exemples. La figure 4-1 illustre la hiérarchie des librairies de X-Window.

Si seule une interface utilisateur élémentaire doit être implémentée, il est possible d'utiliser X Toolkit. Si elle est plus complexe, il faudra probablement utiliser Athena ou Motif. Par ailleurs, Xlib offre les primitives de base pour des graphiques en deux dimensions. Si des extensions 3D sont nécessaires, les librairies PHIGS ou PEX peuvent être utilisées.

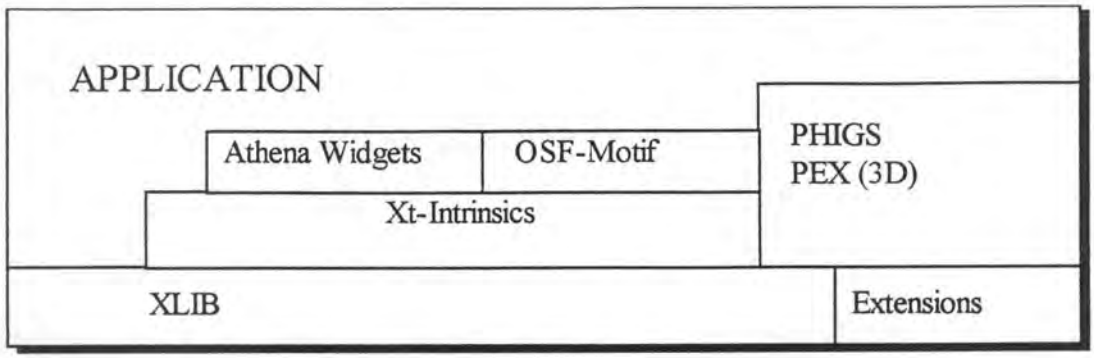


Figure 4–1 : Hiérarchie des librairies de X Window

4.2.2.3 La librairie Xt-Athena

La conception du Whiteboard est basée sur l'utilisation de la librairie Xt et des Widgets de la librairie Athena. Les raisons suivantes justifient ce choix :

- Xt-Athena est libre de tout droit, il est donc aisé de distribuer les applications qui l'utilisent.
- La documentation est relativement aisée à obtenir.
- Xt-Athena est l'ensemble de Widgets le plus simple, il est donc facile de s'en servir comme base. Le code généré est donc aussi probablement plus compact...

4.3 INTERFACE HOMME-MACHINE

4.3.1 INTRODUCTION

Avant d'avancer dans l'analyse et la critique de l'interface du Whiteboard, nous voudrions insister sur deux éléments :

- Le Whiteboard est un outil de haut niveau du bloc multimédia, il n'est donc pas, a priori, une application à part entière, d'autres outils de haut niveau peuvent lui être adjoints (son, image animée).
- Les fonctionnalités du Whiteboard ne sont pas encore totalement fixées, par exemple la gestion des utilisateurs. Cet aspect est important puisqu'il implique entre autres l'existence d'une nouvelle fenêtre.

4.3.2 DESCRIPTION DE L'INTERFACE

Le mode d'utilisation le plus simple du Whiteboard est l'écran passif. Dans ce cas, l'interface se compose d'une simple fenêtre dans laquelle peuvent être affichés :

- une image immobile,
- un pointeur (fixe, en mouvement, masque de fenêtre),
- du texte,
- des tracés géométriques.

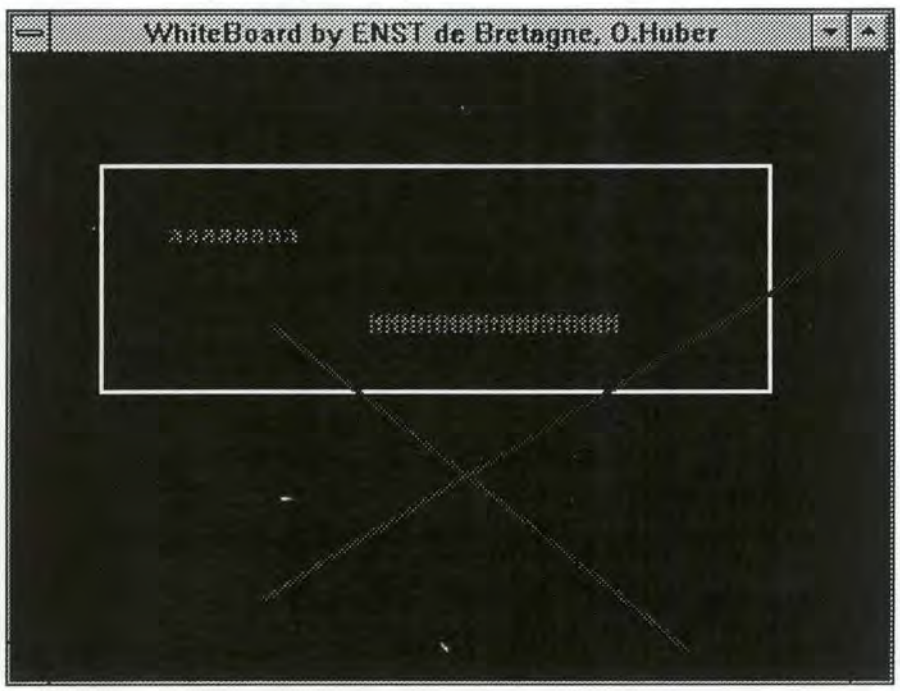


Figure 4–2 : La fenêtre principale du Whiteboard

Le Whiteboard peut également être utilisé pour produire des données. L'écran actif est composé de l'écran passif (figure 4-2) pour afficher les informations et des outils nécessaires à la production des données. Dans la version actuelle de l'application, un poste unique dispose d'un écran actif et produit des données pour un nombre indéterminé de postes disposant d'un écran passif. C'est l'analyse du contenu de la ligne de commande au sein du code de l'application qui détermine le mode.

Dans le cas d'un écran actif, les outils sont répartis en quatre fenêtres.

4.3.2.1 Outils de dessin

La fenêtre des outils de dessin contient six boutons qui déterminent le type de tracé que l'utilisateur est susceptible d'effectuer sur son propre écran et d'émettre : texte, cercle, ligne, rectangle, dessin à main levée et capture d'écran.

4.3.2.2 Couleurs

Cette fenêtre contient dix boutons qui permettent de sélectionner la couleur à utiliser pour les tracés.

4.3.2.3 Pointeurs

Trois boutons permettent de sélectionner les trois types de pointeurs définis dans l'application : un pointeur fixe, un pointeur qui suit les mouvements de la souris et un pointeur pour déterminer la portion de l'image à masquer.

4.3.2.4 Slides

Cette quatrième fenêtre contient l'ensemble des outils permettant d'activer les slides (images préenregistrées). Une liste d'icônes représentant chaque image en taille réduite peut être visualisée permettant la sélection des images par l'utilisateur.

Une barre de défilement verticale permet d'activer les icônes. La sélection de l'icône entraîne son affichage dans la fenêtre principale. Deux boutons situés au-dessus de l'icône permettent respectivement de sélectionner l'icône précédente et l'icône suivante. Un bouton d'effacement est encore introduit.

4.3.2.5 Gestion des utilisateurs

La gestion des utilisateurs ne fait pas encore partie des fonctionnalités du Whiteboard. Les scénarios d'utilisation sont toujours en discussion.

Un scénario possible est que le poste en mode maître dispose d'une fenêtre supplémentaire où sont affichés les noms des personnes désirant prendre la parole. En sélectionnant la personne dans la liste, les outils de tracé ou de sélection sont temporairement actifs sur ce poste.

Un grand nombre de possibilités de partage de l'espace commun sont possibles, le chapitre 3, consacré à l'examen d'exemples d'applications distribuées, en illustre quelques-unes.

4.3.3 CRITIQUES ET PROPOSITIONS D'UTILISATEURS

Le Whiteboard a été utilisé en tant que support d'une application de télé-séminaire dans le cadre d'un travail de fin d'étude en agronomie à L'ENSAR. Ce travail comportait, entre autres, le test de l'outil logiciel dont le Whiteboard faisait partie.

Nous n'insisterons pas sur les remarques faites sur l'usage d'outils informatiques en amont (préparation des slides) ni sur les problèmes de synchronisation entre le son et l'image.

Du point de vue du Whiteboard, nous retiendrons les critiques et propositions suivantes :

- Il est intéressant, du point de vue du conférencier, de réserver les mouvements de la souris aux opérations de tracé ou de déplacement du curseur. L'activation des slides par le clavier est proposée comme solution.
- Le bouton d'effacement est à isoler nettement des boutons d'activation des slides, de façon à éviter les fausses manoeuvres.
- Les icônes d'outils sont à agrandir afin d'éviter à avoir à "viser" lors de chaque utilisation. Un doublement des dimensions est suggéré.
- Deux nouveaux modes d'utilisation sont proposés. Ils définissent l'ensemble des outils dont dispose l'utilisateur. Avec le mode "diaporama ou présentation", l'utilisateur disposerait de la fenêtre d'activation des slides ainsi que des outils de pointage. Avec le mode "rétroprojection ou transparent", l'utilisateur disposerait en plus des outils d'édition. Il est proposé également de définir une version simplifiée pour "débutant" où l'usage de la souris serait paramétré par défaut. Les paramètres seraient les suivants :
  - ◊ pas d'action sur le bouton de la souris pointeur mobile,
  - ◊ un clic sur le bouton gauche : pointeur fixe,
  - ◊ bouton maintenu enfoncé : dessin à main levée,
  - ◊ double-clic : origine de ligne et fin de ligne,
  - ◊ clic sur le bouton droit : accès à un menu de fonctionnalités.

4.3.4 ERGONOMIE DE L'INTERFACE DU WHITEBOARD

4.3.4.1 Critères ergonomiques

La conception d'une bonne interface homme-machine devrait être basée sur le respect d'un certain nombre de règles ergonomiques. Ces règles ont pour mission de respecter un ou plusieurs critères ergonomiques. Le respect de ces critères devrait mener à la conception d'une interface élaborée, efficace, sophistiquée plus conviviale et moins encline à l'erreur.

Huit critères ergonomiques généraux peuvent être définis :

- La compatibilité : Elle concerne la recherche de la cohérence avec l'environnement extérieur à l'application.
- La cohérence : *"Une interface homme-machine est qualifiée de cohérente si, et seulement si, les données et les actions sont facilement identifiables, reconnaissables et utilisables"* (J. VANDERDONCKT, 11). Cette cohérence peut être envisagée au sein d'une application unique ou entre diverses applications.
- La charge de travail : *"Une interface homme-machine est qualifiée d'efficace en charge de travail si, et seulement si, le volume de données à manipuler et d'actions à accomplir par unité de tâche est réduit"* (J. VANDERDONCKT, 11).
- L'adaptabilité : Une interface homme-machine est adaptable si, et seulement si, elle peut être adaptée aux caractéristiques de l'utilisateur (expérience, niveau d'instruction, habitude de travail, langue...).
- Le contrôle du dialogue : Une interface homme-machine est qualifiée d'interface à contrôle explicite si l'utilisateur a l'impression que l'application est sous son contrôle, le but étant de rendre le contrôle de dialogue le plus explicite possible.
- La représentativité : *"Une interface homme-machine est d'autant plus représentative que les codes et libellés utilisés facilitent l'encodage ou la rétention"* (J. VANDERDONCKT, 11).
- Le guidage : Une interface utilisateur est qualifiée d'efficace en guidage si, et seulement si, elle informe de manière constante l'utilisateur sur l'issue de ses actions et sur sa position dans l'accomplissement de la tâche.
- La gestion des erreurs : *"Une bonne interface homme-machine est robuste aux erreurs commises par l'utilisateur et se montre conviviale pour leur correction"* (J. VANDERDONCKT, 11).

4.3.4.2 Critiques ergonomiques

L'ensemble des fenêtres de l'application devrait être regroupé dans une seule fenêtre parent, ceci de façon à *"...rendre plus explicite dans sa présentation les dépendances et les interrelations entre les fenêtres"* (J. VANDERDONCKT, 11). Par ailleurs, ceci supprimerait un autre défaut lié à l'environnement XWindow qui est la fin prématurée de l'application lorsque l'utilisateur ferme une quelconque fenêtre de l'application.

La présentation de la boite de dialogue d'activation des slides pourrait être améliorée. Indépendamment de la position du bouton d'effacement, le rôle respectif des boutons disposés horizontalement (activation des slides) et horizontalement (activation des icônes) n'est pas intuitif. Une règle intéressante à suivre est que *"les boutons de commande de navigation*

doivent suivre une métaphore"(J. VANDERDONCKT, 11). Pourquoi dans le cas qui nous occupe ne pas prendre l'image d'une visionneuse de diapositives ? Pourquoi aussi ne pas ramener les boutons d'activation des slides dans la fenêtre principale et isoler uniquement la gestion des icônes?

Le résultat du dimensionnement des fenêtres n'est pas cohérent : la modification de la fenêtre des outils de dessin n'entraîne pas la modification des boutons radio qu'elle contient alors que c'est le cas lors du dimensionnement de la fenêtre de choix des couleurs.

La gestion des erreurs lors du lancement de l'application n'est que fort tardif, notamment en ce qui concerne la validation des paramètres adresse et numéro de port/flot. Ne peut-on prévoir le choix de ces paramètres sous la forme d'une sélection dans une liste de sélection? Remarquons que ceci n'est pas nécessairement à définir au niveau du Whiteboard, ce pourrait l'être au niveau d'une interface globale à concevoir au niveau de l'application téléséminaire ou de toute autre application utilisant le Whiteboard.

Par ailleurs, avant d'aller plus loin dans la modification de l'interface, rappelons que l'application est destinée à être utilisée dans un environnement hétérogène. Il serait donc intéressant d'examiner les possibilités des environnements physiques les plus courants (Microsoft Windows, Apple MacIntosh, OSF-Motif ...) de façon à concevoir une interface la plus cohérente possible.

4.3.5 MODIFICATIONS

Les modifications que nous avons introduites sont présentées dans les points suivants.

4.3.5.1 La taille des outils de dessin et de pointage

La taille des outils de dessin peut, maintenant être modifiée proportionnellement à la taille de la fenêtre parent (figure 4-3). Cette modification implique un changement dans le type des Widgets utilisées pour les boutons et dans les ressources définissant leurs positions relatives.

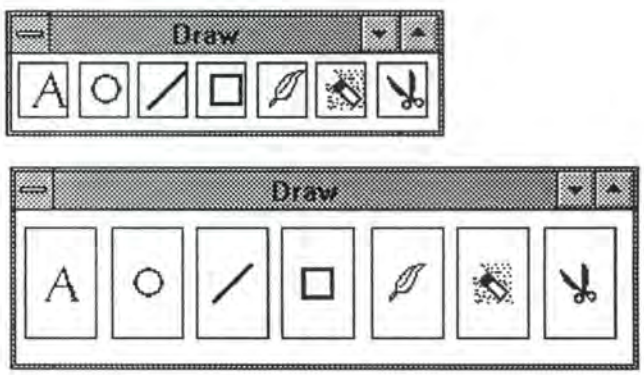


Figure 4–3 : Les outils de dessin du Whiteboard

La taille des outils de pointage est modifiée de façon similaire (figure 4-4).

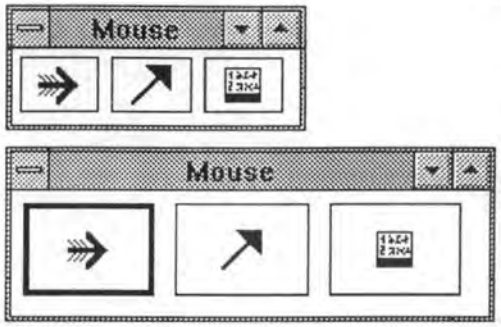


Figure 4–4 : Les outils de pointage du Whiteboard

4.3.5.2 La position du bouton d'effacement de la fenêtre des "slides"

Le bouton d'effacement de la fenêtre principale se trouve maintenant dans la partie inférieure de la fenêtre slides. Cette modification implique la modification des ressources définissant la position du bouton d'effacement par rapport aux autres "Widgets" présentes dans la fenêtre. Cette fenêtre est également redimensionnable (figure 4-5).

Ces modifications ont été introduites dans le code de l'application. Cette solution est préférable à celle qui consiste à utiliser un fichier de ressources externe modifiable par l'utilisateur car la manipulation de ces ressources n'est pas particulièrement aisée.

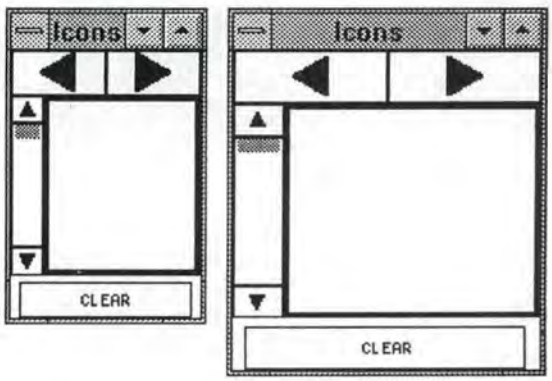


Figure 4–5 : La fenêtre d'activation des slides

4.3.5.3 Introduction de "raccourcis clavier"

Des "raccourcis clavier" sont maintenant disponibles pour toutes les fonctions de la fenêtre des "slides". Du point de vue de l'implémentation, cette modification a été réalisée en ajoutant une nouvelle fonction au Whiteboard pour assurer la gestion des événements clavier :

```
void wbSlideKeyEvent (      Widget          focusWidget,
                            XtPointer       client_data,
                            XKeyEvent       *keyevent )
```

La fonction `XtAddEventHandler()` de la librairie Xt dont la syntaxe est reprise ci-dessous permet d'associer l'appel de cette fonction à un événement clavier et à une Widget

déterminée. Remarquons que la fonction définie par le programmeur sera appelée pour la Widget passée comme paramètre mais également pour chacune de ses filles.

```
void XtAddEventHandler(     Widget          w,
                            XtEventMask     event_mask,
                            Boolean         non_maskable,
                            XtEventHandler  proc,
                            XtPointer       client_data)
```

4.3.5.4 Perspectives

Les autres modifications proposées par les utilisateurs devraient être étudiées avec plus de soin car elles impliquent un choix de la part de l'utilisateur. De façon plus générale, il convient de définir le moyen de paramétrer au maximum l'interface de l'application et ceci à deux niveaux :

1. par l'utilisateur final,
2. par la personne chargée de mettre au point l'application multimédia.

Cette paramétrisation devrait être dynamique pour l'utilisateur final. Dans ce cas, il y a plusieurs possibilités : la première est d'ajouter un outil de contrôle interne au Whiteboard (boite de dialogue, menu déroulant...), la seconde de définir un contrôle externe à l'outil qui pourrait être supporté par le mécanisme de communication entre clients X. L'outil de contrôle devrait être défini au niveau de l'application téléséminaire ou de toute autre application utilisant le Whiteboard. L'avantage de cette solution serait de pouvoir contrôler avec une interface commune tous les outils de haut niveau d'une application (son, image animée...).

La paramétrisation serait statique pour le développeur. Dans ce cas, elle pourrait être réalisée par des options de la ligne de commandes à définir. Sachant que la manipulation des ressources de X-Window est relativement délicate, les options devraient définir des paramètres plus généraux que les ressources standard de la librairie Xt ou Athena en définissant un nombre limité de configurations (nombre et disposition d'outils).

4.4 GESTION DE LA COULEUR ET DES FORMATS D'IMAGE DANS LE WHITEBOARD

4.4.1 INTRODUCTION

Un des objectifs du projet PRISM est de fournir des outils pour le développement d'applications multimédia dans un environnement hétérogène. La gestion de la couleur est un des domaines où accepter l'hétérogénéité rend délicate la conception d'une application. En effet, une conception générique de la gestion de la couleur est impossible si l'on désire utiliser chaque environnement au maximum de ses possibilités.

Après avoir présenté brièvement la diversité du matériel auquel peut être confronté le développeur d'application, nous verrons comment cette diversité a été prise en compte dans la version initiale du Whiteboard. Après avoir mis en évidence les limites de ces choix, nous montrerons comment nous avons pu améliorer l'application initiale.

Par ailleurs, le seul format de fichier d'image que la version initiale du Whiteboard accepte est le format GIF. La version modifiée accepte le format PCX, format très courant dans le monde PC. Le mécanisme de conversion est simple et peut aisément être étendu à d'autres formats de fichiers graphiques.

4.4.2 VARIETE DES ECRANS

Il existe une grande variété d'écrans. Malheureusement, du point de vue de l'analyste ou du programmeur, il n'existe pas une seule stratégie pour gérer l'ensemble des affichages.

Actuellement, de nombreuses stations de travail disposent d'écrans 256 couleurs. Parmi ceux-ci, un grand nombre travaille en pseudo-couleurs. On parle de pseudo-couleurs car la valeur de chaque pixel est un indice dans une table de couleurs et non une couleur. D'autres encore travaillent en couleurs véritables car le nombre de couleurs affichables en même temps peut atteindre plus de 16.10^6 couleurs; dans ce cas, les palettes de couleurs peuvent être modifiables ou non. Certains postes de travail sont équipés d'écrans qui n'acceptent que des niveaux de gris dont le nombre peut varier. Enfin, de nombreux postes de travail sont encore équipés d'écrans noir et blanc.

4.4.3 GESTION DE LA COULEUR DANS L'ENVIRONNEMENT XWINDOW

4.4.3.1 Modélisation de la couleur

X Window offre plusieurs modèles pour représenter la couleur :

Suivant le modèle RGB, chaque couleur est définie par un triplet (R,G,B) dont les valeurs donnent les intensités des trois couleurs fondamentales. Ces intensités sont directement traduites en termes de tension électrique à appliquer aux canons à électrons de l'écran. Cette méthode pose au moins deux problèmes :

1. La représentation est fortement dépendante du matériel.
2. La perception humaine de la couleur n'est pas directement proportionnelle à la tension appliquée à chacun des trois canons à électrons de l'écran.

Lorsque ces deux aspects sont déterminants dans la conception d'une application, une autre représentation, basée sur un standard international, peut être utilisée. Le modèle CIEXYZ permet la spécification des couleurs indépendamment du matériel. Dans ce modèle, une couleur est décrite par la valeur de trois coordonnées dont l'interprétation est malheureusement loin d'être intuitive.

X11R5 supporte également le modèle TekHVC. La représentation des couleurs est proche de celle du modèle précédent mais plus intuitive. Dans ce modèle, une couleur est représenté par sa chromaticité qui définit la saturation, la teinte qui définit la couleur spectrale et la valeur qui définit la l'intensité de la couleur.

La conversion de ces couleurs en intensité RGB est particulière à chaque écran et est réalisée par un client X particulier (xcmsdb).

4.4.3.2 Gestion de la couleur

La gestion des couleurs dans X Window repose sur des palettes de couleurs ou tables de couleurs. D'un point de vue conceptuel, une palette de couleur est une table de N cellules, chaque cellule étant composée de trois champs déterminant les intensités RGB. Pour afficher un point à l'écran, l'application met une certaine valeur dans la mémoire écran du système. Si on associe un bit à chaque pixel, on obtient un plan écran, en multipliant le nombre de plans, on multiplie le nombre de couleurs affichables à un moment donné. Le nombre de couleurs utilisables en même temps est déterminé par la relation couleurs = 2^{nplans}. Le principe de l'utilisation des palettes de couleurs est illusté par la figure 4-6.

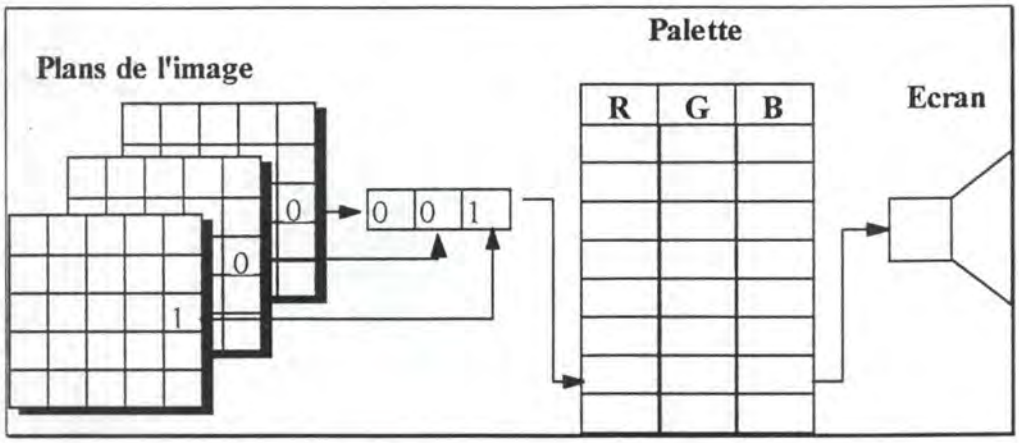


Figure 4–6 : Les palettes de couleurs

Les applications courantes ne font que manipuler les indices de tables de couleurs définies par défaut. Cependant, à chaque fenêtre X peut être associée une palette virtuelle. Lorsque la fenêtre est activée, les couleurs contenues dans cette palette sont alors physiquement installées. Par convention, ce sont les gestionnaires de fenêtres qui installent la palette associée à la fenêtre active. Si nous prenons le cas d'un écran qui autorise l'affichage simultané de 256 couleurs, nous avons un système qui permet d'ajuster la palette de couleurs aux besoins spécifiques de chaque image et d'étendre virtuellement le nombre de couleurs disponibles. L'inconvénient majeur est que toutes les autres fenêtres de l'écran apparaissent en fausses couleurs.

X Window introduit la notion de « visual », ce terme difficile à traduire définit la manière dont peut être gérer un écran particulier. Remarquons dès à présent que, souvent, plusieurs visuals conviennent à un écran. Ce visual se présente concrètement sous la forme d'une structure de données définie dans Xlib.

Normalement, le programmeur n'accède pas directement à cette structure, celle-ci est en effet sujette à des modifications au cours des mises à jour de X11. Pour accéder aux informations contenues dans les visuals, plusieurs fonctions comme `XgetVisualInfo()` renvoient une structure `XVisualInfo` qui, elle, n'est pas sensée devoir évoluer. Le champ classe de cette structure contient une constante caractérisant une classe de visual. Cette constante définit la façon principale d'utiliser un écran, elle peut prendre les valeurs suivantes : DIRECTCOLOR, GRAYSCALE, PSEUDOCOLOR, STATICCOLOR, STATICGRAY et TRUECOLOR. Le tableau 4-1 compare les différentes classes de visual et montre les opérations qui peuvent être faites sur la table de couleurs. Certains écrans peuvent supporter plusieurs types de visual. La figure 4-7 illustre la hiérarchie des visuals.

Tableau 4-1 : visuals et opérations possibles

Type de table de couleurs	Ecriture et lecture	Lecture seulement
Monochrome/ niveaux de gris	GRAYSCALE	STATICGRAY
Index unique (RGB)	PSEUDOCOLOR	STATICCOLOR
Index séparé (RGB)	DIRECTCOLOR	TRUECOLOR

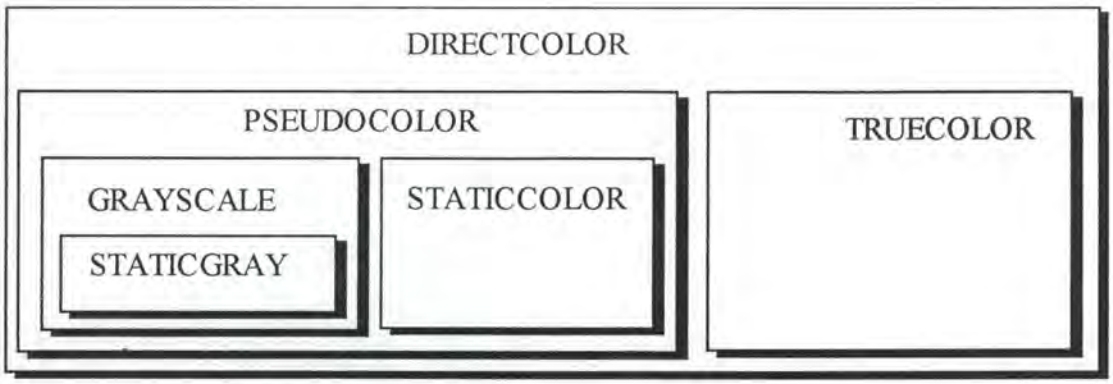


Figure 4–7 : Hiérarchie des "visuals"

4.4.4 IMAGES ET PIXMAPS

Une fenêtre qui est affichée à l'écran peut être masquée partiellement ou encore voir sa taille modifiée. Selon les conventions définissant le rôle du gestionnaire de fenêtre, la mise à jour du contenu des fenêtres n'est pas de sa responsabilité, c'est l'application qui en fonction des événements qu'elle aura recu prendra les mesures appropriées. Par ailleurs, le contenu de la fenêtre doit être conservé.

Il y a trois solutions sous X-Window pour stocker des graphiques.

1. Ils peuvent être stockés directement dans la fenêtre mais la fenêtre peut être recouverte, d'où une perte de données. Le graphique doit donc être conservée ailleurs.
2. Le contenu de la fenêtre peut également être conservé dans la mémoire du client sous la forme d'une structure d'image.
3. Le contenu de la fenêtre peut être conservé dans la mémoire du Serveur X sous la forme d'un pixmap.

L'avantage de conserver les données dans une image est la taille théoriquement illimitée que celle-ci peut avoir. Par contre, il n'existe aucune routine d'usage simple pour gérer les graphiques dans une image (tracé une ligne, un cercle...). De plus, lorsque le contenu de la fenêtre doit être mis à jour, cela implique des transferts de données importants sur le réseau.

L'utilisation de pixmap répond à ces deux objections: à la première car il est possible d'utiliser des routines graphiques dans une fenêtre comme dans un pixmap, ces deux "destinations" d'une routine graphique sont reprises sous le terme de "drawable". Par ailleurs, le pixmap est conservé au niveau du serveur, ce qui limite la charge du réseau. L'inconvénient majeur de l'utilisation de pixmap est que l'espace mémoire est généralement plus restreint. La figure 4-8 montre les relations qui existent entre les trois formes de support aux images ainsi que les primitives qui permettent les transferts d'un support vers un autre.

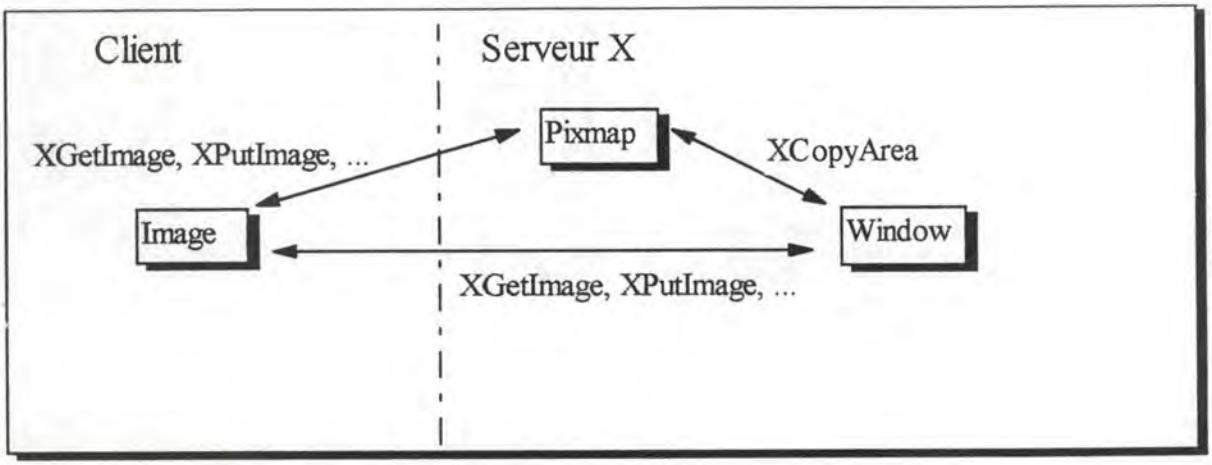


Figure 4–8 : Gestion des images sous X-Window

4.4.5 LA COULEUR DANS LE WHITEBOARD

4.4.5.1 Spécifications

La version initiale du Whiteboard est conçue pour n'utiliser que des écrans de type pseudo-couleur. Ce choix peut être justifié par :

- L'abondance de ce type d'écran.
- Le fait que le Whiteboard doit garantir une bonne qualité des images présentées sur l'ensemble des postes "esclaves". Il est donc souhaitable que les couleurs soient les mêmes pour tout le monde, ce qui est relativement facile à réaliser avec ce type d'écran car :
    1. à chaque élément de la table sont associées des intensités de rouge, de vert et de bleu,
    2. ces trois intensités contrôlent directement le moniteur,
    3. le nombre de couleurs affichables sur un écran (16,7 millions) est nettement supérieur au nombre de couleurs affichables en même temps.

Ce choix pose cependant plusieurs problèmes :

- La volonté d'afficher des images de qualité et identiques nécessite l'usage d'une table de couleurs privée (propriété de l'application). Cette table de couleurs virtuelle est chargée dans la table physique lors de chaque activation de l'application, le restant de l'écran apparaît en conséquence en fausses couleurs, ce qui peut être désagréable.
- Un bon nombre d'écrans et de cartes vidéo n'autorisent pas la création de tables de couleurs virtuelles.
- La gestion des écrans monochromes et de type niveaux de gris n'est pas prise en compte.

Par ailleurs, le Whiteboard est un des outils de base pour des services multimédia de haut niveau. Il est préparé pour coopérer et partager les ressources avec d'autres clients X. Pour cette raison, une part de la table de couleurs est réservée pour de futures applications vidéo. La figure 4-9 montre le partage de la table de couleurs tel qu'il est réalisé dans l'application.

Le Whiteboard utilise 128 couleurs pour les slides au lieu des 256 disponibles, ce qui permet avec le codage GIF le partage de la table de couleurs et la réduction de la taille des fichiers dans lesquels les slides sont conservés.

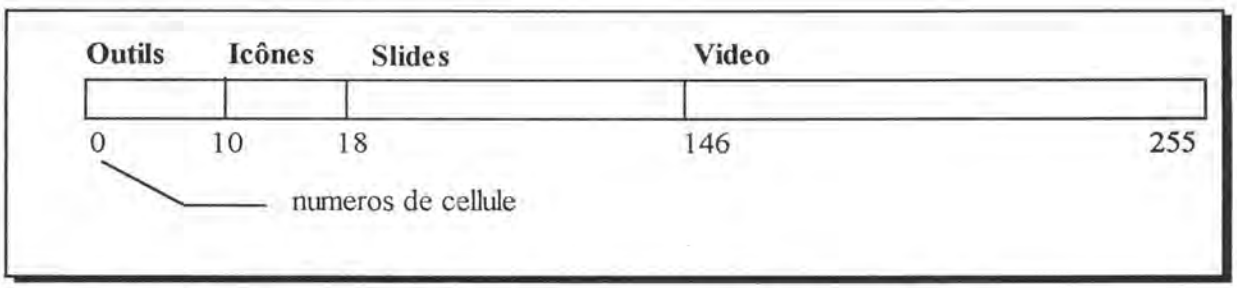


Figure 4–9 : Partage de la table de couleurs du Whiteboard

4.4.5.2 Implémentation de la version initiale du Whiteboard

Avant d'exposer les modifications introduites, il convient ici d'expliquer certains détails de l'implémentation. Les quelques fonctions courantes de X11 pour la manipulation des tables de couleurs utiles à la compréhension de l'implémentation seront présentées avant d'examiner l'application de façon plus approfondie.

a. QUELQUES FONCTIONS DE X11

```
Colormap XCreateColormap(      Display   *display,
                               Window    w,
                               Visual    *visual,
                               int       alloc      )
```

La fonction `XCreateColormap()` crée une table de couleurs pour la fenêtre window associée au passé comme premier paramètre et adaptée au visual passé comme troisième

paramètre. Le dernier paramètre est une constante qui prend la valeur `allocAll` ou `allocNone`. Dans le premier cas, la table de couleurs entière est allouée en écriture. Remarquons que la valeur initiale des entrées allouées est indéterminée.

```
XAllocColorCells(    Display *display,
                     Colormap colormap,
                     Bool contig,
                     unsigned long plane_masks_return[],
                     unsigned int nplanes,
                     unsigned long pixels_return[],
                     unsigned int npixels     )
```

La fonction `XAllocColorCells()` alloue un ensemble de "cellules" de couleurs. Le paramètre `contig` permet de préciser si les plans retournés par le paramètre `plane_masks_return` doivent être contigus. Ce paramètre sert à masquer certains plans. Le paramètre `nplanes` détermine le nombre de plans utilisés. Ce paramètre peut valoir 0 si l'application ne doit pas gérer chaque plan indépendamment, ce qui est généralement le cas. Le paramètre `pixel` contient un pointeur sur les cellules allouées. Le paramètre `npixels` détermine le nombre de cellules à allouer.

```
XStoreColors(   Display   *display,
                Colormap  colormap,
                XColor    color[],
                int       ncolors    )
```

La fonction `XStoreColors()` change les entrées de la table de couleurs à partir des valeurs contenues dans les structures `XColor` passées comme paramètres. Pour que les valeurs soient modifiées les flags `DoRed`, `DoBlue` et `DoGreen` (constantes définies dans Xlib) réunis par l'opérateur OR doivent être placés dans le champ `flag` de la structure. Le champ `ncolors` spécifie le nombre de couleurs présentes dans le tableau `color`.

b. CREATION D'UNE TABLE DE COULEUR ET POLITIQUE D'UTILISATION

La fonction `wbInitializeColormap()` du Whiteboard a pour objet de créer une table de couleurs associée à la fenêtre racine de l'écran et d'allouer de l'espace en mémoire pour les cellules de la palette. Des couleurs prédéfinies sont affectées aux dix premières cellules de la table pour les outils de dessin.

La fonction `wbLoadColorMap()` a pour objet de charger la palette de couleurs de l'image (slide) dans la table de l'application. Puisque cette table doit pouvoir être partagée (voir ci-dessus), les couleurs sont chargées dans la table avec un offset déterminé et fixe.

c. GESTION DES AFFICHAGES DES CURSEURS ET UTILISATION DE CONTEXTE GRAPHIQUE

Trois types de curseur sont utilisés par l'application. Le premier est destiné à marquer de façon fixe un élément de l'image. Le second est mobile, il suit les mouvements de la souris du poste "maître". Le troisième est destiné à limiter la zone d'affichage de l'écran. Le problème qui est posé est de réaliser l'affichage des curseurs tout en conservant l'image sous-jacente mais également de limiter le volume des informations à véhiculer sur le réseau.

Le second problème est résolu aisément en utilisant des pixmaps (tableau de pixels résidant en mémoire au niveau du serveur) pour l'image et pour le tracé du curseur. Les seules informations qui doivent être transmises sont les coordonnées du curseur dans la fenêtre.

La solution au premier problème nécessite plusieurs manipulations : pour chaque curseur deux pixmaps sont créés. Dans le premier, tous les pixels ont une valeur de 255 (tous les bits sont à 1) à l'exception du tracé du curseur proprement dit où les pixels ont une valeur de 0.

Le résultat de l'opération "ET" sur ces deux pixmaps donne une image où l'image initiale est conservée à l'exception du tracé du curseur qui est entièrement blanc. Un deuxième pixmap du curseur est défini où l'ensemble des pixels ont la valeur 0 à l'exception du tracé du curseur, tracé légèrement en retrait par rapport au premier.

En appliquant l'opération "OU" sur ce pixmap et le résultat de l'opération précédente, on obtient le tracé du curseur noir avec un fin liseré blanc tout en maintenant l'image inchangée.

Les pixmaps des curseurs sont créés à l'aide de la fonction de Xlib `XCreatePixMapFromBitmapData()` qui permet de définir les valeurs à associer à l'avant-plan et à l'arrière-plan. La fonction `XCopyArea()` copie un premier pixmap sur un deuxième à une position passée comme paramètre. L'opération logique utilisée lors de la copie est précisée à l'aide d'un contexte graphique passé comme paramètre. Le contexte graphique est créé à l'aide de la fonction `XCreateGC()`. La succession des opérations effectuées est illustrée par la figure 4-10.

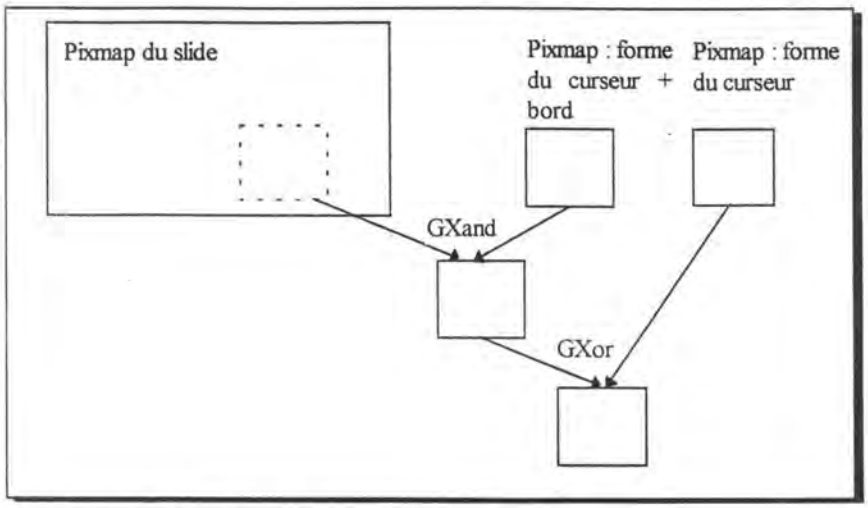


Figure 4–10 : Gestion des outils de pointage

4.4.6 MODIFICATIONS DE LA GESTION DE LA COULEUR

L'objectif est d'élargir la gamme des écrans supportés par l'application.

Il nous a été demander de modifier l'application afin qu'elle supporte les écrans monochromes, ce qui présente l'avantage que la gestion de la couleur pour un tel type d'écran (visual STATICGRAY) reste valable pour tous les autres.

La gestion de la couleur comporte deux parties :

- la gestion locale (dégradation des images couleurs, gestion des couleurs des outils de dessin),
- la gestion de la communication (dégradation des messages liés à la couleur, dans le sens maître-esclave et esclave-maître).

4.4.6.1 Gestion locale

a. LE DITHERING

Le dithering d'une image couleur donne une image noir et blanc avec un effet d'image en niveau de gris. L'algorithme utilisé ici est celui de Floyd-Steinberg.

b. PRINCIPE DE L'ALGORITHME

Le contenu de la table de couleurs de l'image est transformé en niveaux de gris (cette transformation est faite au début de l'algorithme si la taille de la table n'est pas trop grande). Chaque niveau de gris est obtenu en utilisant la somme pondérée des valeurs RGB pour chaque des entrées de la table. Ces niveaux de gris sont ensuite ajustés de façon à ne pas dépasser un certain seuil.

Le dithering proprement dit est effectué ligne par ligne. D'abord de droite à gauche puis de gauche à droite à la ligne suivante. Cette opération est effectuée par deux fonctions: `RightToLeft()` et `LeftToRight()`. Chaque pixel de chaque ligne est comparé à un

seuil, tout pixel dont la valeur est inférieure à ce seuil se verra affecté d'une valeur maximale ou minimale dans le cas contraire. L'erreur (écart entre le seuil et la valeur affectée) est répartie sur les pixels voisins.

c. IMPLEMENTATION DU DITHERING

Nous sommes partis d'une implémentation existante de l'algorithme de dithering(Xloadimage, 13) La principale difficulté de cette phase a été d'intégrer le code existant avec ses structures de données propres dans le Whiteboard tout en conservant la cohérence du programme initial. A cette occasion, deux fonctions du Whiteboard ont été modifiées: `wbShowSlide()` et `wbShowIcon()`. Ces fonctions ont pour objet d'afficher respectivement un slide et une icône. Elles font toutes deux appel à la fonction `LoadGif` du module gifread pour charger en mémoire le contenu d'un fichier gif. Cette fonction qui fait appel à des variables globales a été modifiée de façon à retourner une structure de type image utilisée par la fonction de dithering. Nous avons préféré cette solution à celle qui consistait à modifier complètement le code de la fonction de dithering `dither` et à utiliser les variables globales. Lorsque l'image est chargée en mémoire, le test de la variable `BlackWhite` permet de déterminer si le type de visual nécessite un dithering. Dans ce cas, la fonction `dither()` est appelée.

Deux autres difficultés doivent être mentionnées ici :

1. Dans le cas d'un écran noir et blanc ou assimilé, lorsqu'un fichier gif contient une image monochrome, il ne faut appliquer aucune opération de dithering sur cette image. La fonction de dithering `dither()` a donc été modifiée de façon à tester le champs `depth` (nombre de bits par pixel) de la structure image, le dithering n'est effectué que si ce champ est strictement supérieur à un.
2. L'ordre des bits peut varier d'une machine à l'autre et il n'est pas nécessairement le même sur la machine où s'exécute l'application que sur celle où se trouve le "display". Pour les pixmaps et les bitmaps, l'ordre des bits est défini par le serveur. Les clients dont l'ordre est inverse doivent réaliser eux-mêmes la conversion. Remarquons que pour toute autre partie du protocole, le serveur réalise l'inversion automatiquement. Le problème s'est présenté lors de la phase de compactage de l'image après le dithering. Cette phase est réalisée par des opérations de décalage au niveau des bits. Les macro `ImageByteOrdrer()` et `BitmapBitOrder()` permettent de déterminer respectivement l'ordre des octets et des bits utilisés par le serveur.

4.4.6.2 Gestion de la communication

Lorsqu'un poste monochrome ou assimilé reçoit une information de type couleur, celle-ci est automatiquement interprétée comme étant du noir, à l'exception du blanc.

Lorsqu'un poste monochrome ou assimilé envoie une information de type couleur, celle-ci ne peut être que du blanc ou du noir.

Les écrans de type niveaux de gris (classe de visual GRAYSCALE et STATICGRAY) sont ainsi utilisés en dessous de leurs possibilités.

4.4.7 FORMATS D'IMAGE DANS LE WHITEBOARD

Le seul format d'image que reconnait la version initiale du Whiteboard est le format GIF. Rappelons que le choix de ce format peut se justifier par les possibilités de manipulation aisée de ce format dont la réduction du nombre de couleurs. Par ailleurs, la conversion d'une image GIF en une structure Ximage définie et manipulée par Xlib est relativement aisée.

4.4.8 EXTENSION DU NOMBRE DE FORMATS

L'objectif est de pouvoir accepter de nouveaux formats de fichier sans modifier fondamentalement la structure de l'application et sans ralentir le chargement des images lors de l'exécution de l'application.

La solution implémentée est la suivante : lors du démarrage de l'application, au moment du chargement de la liste des slides par la fonction `wbGetSlideList()`, une nouvelle fonction est appelée :

```
void wbCheckFormat(char  *currentSlide)
```

Cette fonction analyse le suffixe du nom du fichier passé comme paramètre et, dans le cas où ce suffixe correspond à un format reconnu, elle fait appel à une autre nouvelle fonction :

```
void wbConvert(char *oldName, char *script)
```

Cette fonction reçoit comme paramètre une chaîne de caractères contenant le nom du fichier à convertir ainsi que le nom du script de conversion à utiliser. Un processus fils est créé par l'appel `fork()`, le nom du script est concaténer avec les autres paramètres nécessaires et son exécution est lancée par un appel `system()`. Le processus parent (le whiteboard) attend alors la fin de la conversion par un appel `wait()`. Seule la conversion de fichier au format pcx est ainsi actuellement exécutée, la reconnaissance des fichiers au format postscript est également assurée mais aucun script n'a encore pu être mis au point.

Une fois le fichier converti, le suffixe ".conv" est rajouté au nouveau fichier et s'est ce nom qui est conservé dans la liste des slides.

Cette solution ne ralentit donc l'exécution que dans la phase de démarrage de l'application et nullement lors de l'activation des slides. L'ajout de nouveaux formats de fichier est relativement simple à implémenter pour autant que des utilitaires de conversion soient disponibles. Cette modification comprend :

- l'ajout d'un test supplémentaire dans la fonction `wbCheckFormat()`,
- la création d'un script de conversion.

4.5 COMMUNICATION AVEC LE WHITEBOARD

4.5.1 INTRODUCTION

Rappelons que dans l'état actuel le Whiteboard peut être actif selon deux modes non exclusifs (le mode "maître" et le mode "esclave"). La fonctionnalité principale du Whiteboard en mode esclave est de recevoir des paquets en provenance d'un poste maître et d'interpréter leurs contenus en terme d'actions d'affichage à l'écran. En mode maître, cette fonctionnalité est de traduire les actions de l'utilisateur sous la forme de paquets envoyés à un ensemble de postes esclaves.

L'objectif de ce chapitre est de montrer comment ces fonctionnalités sont mises en oeuvre en insistant sur ce qu'offre la plate-forme PRISM au concepteur d'application distribuée.

Les notions de datagramme et de multicast (IP) sont à la base de la conception du bloc de distribution de PRISM, nous envisagerons donc d'abord ces deux notions.

l'API de distribution de PRISM étant construite sur base de la "socket interface", nous présenterons les principes de la programmation avec les sockets de UNIX. Nous poursuivrons par l'analyse de quelques fonctions de l'API de distribution. Nous terminerons enfin par l'analyse de l'utilisation de cette API dans le cadre du Whiteboard.

4.5.2 UDP (USER DATAGRAM PROTOCOL)

Le protocole UDP définit un service de transport sans connection dans un environnement de réseaux interconnectés. Les paquets échangés sont appelés "datagramme". Le protocole IP est le protocole réseau sous-jacent.

UDP offre une procédure à des programmes d'application pour envoyer des messages à d'autres applications avec un protocole minimal. La délivrance et l'absence de duplication des messages ne sont pas garanties à l'opposé du protocole TCP qui rencontre ces exigences. Par rapport au protocole IP, le seul véritable ajout de UDP est la définition d'un port. Ce port permet à plusieurs entités de niveau applicatif de cohabiter sur la même machine.

Toute interface utilisateur pour le protocole UDP offre les services suivants :

- Création de nouveaux ports.
- Réception sur ces nouveaux ports: cette opération retourne les octets de données ainsi qu'une indication du port et de l'adresse d'origine.
- Emission de datagramme en spécifiant les données à envoyer ainsi que les adresses (source et destination) et port (source et destination).

Le protocole UDP est probablement plus adapté au transfert de données multimédia que ne l'est TCP.

Ce dernier protocole garantit l'arrivée des paquets émis et leur réception dans le bon ordre. Il assure donc, le cas échéant leur réémission. Pour les raisons invoquées dans le chapitre 2, les données multimédia doivent arriver dans le bon ordre certes mais la perte d'un paquet est moins fondamentale que l'exigence de pouvoir émettre et recevoir les données à la vitesse où elles sont produites. UDP, en ne réemettant pas les paquets perdus, n'introduit pas de retard supplémentaire. Si une sécurisation partielle du flux est nécessaire, elle peut être réalisée au-dessus de UDP, c'est ce que réalise PTP en mode émulation.

Par ailleurs, TCP est incompatible avec le multicasting comme nous le verrons ci-après.

4.5.3 MULTICASTING ET LE PROTOCOLE IP

Le multicasting est défini dans le protocole IP comme étant la transmission d'un datagramme à un groupe d'hôtes qui peuvent être dispersés sur plusieurs réseaux locaux. Il ne faut pas confondre cette notion avec le broadcasting (envoi d'un message destiné à tous les hôtes sur le réseau ou le sous-réseau).

Un groupe d'hôtes peut être permanent ou temporaire. Lorsque le groupe est permanent, une adresse IP lui est assignée de façon permanente. Lorsque le groupe est temporaire, l'adresse est assignée dynamiquement à la requête d'un des hôtes.

La création et la maintenance de groupes multicast sont de la responsabilité d'agents multicast. Ces entités résident sur les passerelles internet ou sur un hôte affecté à cet usage. Il y a au moins un agent multicast directement attaché à tout réseau IP qui supporte le multicasting internet. Ce sont ces agents qui sont responsables de la retransmission d'un datagramme lorsqu'un membre du groupe fait partie d'un autre réseau. La gestion de ces groupes se fait par l'intermédiaire du protocole IGMP (Internet Group Management Protocol). Notons enfin qu'un hôte peut envoyer un datagramme à un groupe sans en faire partie.

Un groupe multicast est généralement représenté par une adresse IP de classe D. De façon à autoriser une flexibilité maximale, plusieurs classes d'adresse sont définies par le protocole IP. Le champ adresse du paquet est codé de façon à permettre l'adressage d'un

nombre plus ou moins important de réseaux par rapport au nombre d'hôtes dans chaque réseau. Le tableau 4-2 donne les spécifications de chaque classe d'adresse. Ce sont les bits de poids fort du champ d'adresse qui déterminent la classe d'adresse.

Tableau 4-2 : Définition des classes d'adresse IP

Classe	Bits de poids fort	Réseau	Hôte
A	0	7 bits	24 bits
B	10	14 bits	16 bits
C	110	21 bits	8 bits
D	1110	/	29 bits

Pour transmettre un datagramme destiné à un groupe d'hôte, le module IP doit réaliser la correspondance entre l'adresse du groupe de destination et une adresse du réseau local. Comme pour les adresses IP individuelles, l'algorithme qui réalise cette correspondance est spécifique au réseau local. Sur les réseaux qui supportent directement le multicast, l'adresse de groupe IP est mappée directement en adresse multicast du réseau local. Pour les réseaux qui ne le supportent pas, le mapping peut être une adresse de broadcast, une liste d'adresses unicast ou encore l'adresse d'une machine qui va gérer la diffusion.

4.5.4 LE CONCEPT DE SOCKET

De façon à simplifier la conception d'applications utilisant les communications réseau, les concepteurs des sockets ont choisi de définir un support de communication général qui accepte différentes familles de protocole de communication. Plusieurs familles de protocoles peuvent être utilisées : par exemple IP, Xerox NS...

Quand un programme demande de créer un socket, le système d'exploitation renvoie un entier appelé descripteur de socket pour le référencer. Lors de l'appel `socket`, le système ajoute une nouvelle entrée dans la table des descripteurs de sockets, cette entrée contient un pointeur sur une structure de donnée `sockaddr`. Après l'appel, seuls deux champs de cette structure sont remplis : la famille et le service. D'autres appels compléteront les champs vides de la structure.

4.5.4.1 Spécifier une extrémité de connection

Pour permettre à chaque famille de protocoles de définir sa propre représentation des adresses de connexion, les sockets définissent des familles d'adresse. Une famille de protocoles peut utiliser une ou plusieurs familles d'adresses.

Les adresses nécessaires au socket sont stockées dans des structures prédéfinies. La structure la plus générale est la structure `sockaddr`. Elle se compose d'un champ de deux octets pour spécifier la famille d'adresses et de 14 octets utilisés d'une façon spécifique à chaque protocole.

4.5.4.2 Les appels systèmes pour les sockets

Nous ne présenterons ici que les appels utilisés par le Whiteboard et l'API de distribution, ce sont les seuls utiles à la compréhension de la suite de cette étude.

a. L'APPEL SOCKET

L'appel crée un nouveau socket et retourne un descripteur pour celui-ci.

```
int   socket(   int famille,
                int type,
                int protocole  )
```

`famille` détermine la famille de protocoles, par exemple :

`AF_UNIX`	système de fichier UNIX
`AF_INET`	protocole Internet
`AF_NS`	protocole Xerox NS
`AF_IMPLINK`	protocole IMP link layer
`AF_PRISM`	protocole PRISM

`type` détermine le type de service, par exemple :

`SOCK_STREAM`
`SOCK_DGRAM`

b. L'APPEL CLOSE

Quand un programme a fini d'utiliser un socket, l'appel `close` permet de désallouer le socket.

```
int close( sockfd )
```

`sockfd` est le descripteur de socket.

c. L'APPEL SENDTO

Dans le cas d'un socket en mode déconnecté, l'application fait appel à `sendto` pour spécifier à la fois le datagramme à envoyer et l'adresse de destination.

```
int sendto(     sockfd,
                char *message,
                int msglen,
                int  flags,
                struct sockaddr *toaddr,
                int toaddrlen )
```

`sockfd` est le descripteur de socket.

`message` est l'adresse du tampon de donnée à envoyer.

`flags` définit les options de déboguage.

`toaddr` est un pointeur sur la structure contenant l'adresse de destination.

`toaddrlen` précise la longueur de la structure d'adresse.

d. L'APPEL RECVFROM

Lors d'un échange de données en mode non connecté, `recvfrom` peut être utilisé pour recevoir des données. Lors de l'appel `recvfrom`, le processus appelant est bloqué jusqu'à l'arrivée d'un datagramme. Le système place le datagramme arrivé dans un tampon de données et l'adresse de l'envoyeur dans un autre tampon. Cette adresse peut servir, dans le cas d'un serveur, à déterminer l'adresse du client à qui renvoyer la réponse à une requête.

```
int recvfrom (  int sockfd,
                     char *buf,
                     int len,
                     int  flags,
                     struct sockaddr *from,
                     int fromlen     )
```

`sockfd` est le descripteur du socket.

`buf` est l'adresse du tampon pour stocker les données reçues.

`len` est la longueur du tampon de données.

`flags` définit les options de déboguage.

`from` est un pointeur sur une structure pour stocker l'adresse de l'envoyeur.

`fromlen` est la longueur du tampon contenant l'adresse de l'envoyeur.

4.5.5 L'API DU BLOC DE DISTRIBUTION

4.5.5.1 POpenSender()

```
int POpenSender (    char                *address,
                     int                 port,
                     struct sockaddr     **sa,
                     int                 *lg_resp   )
```

Au sein de la fonction, le contenu du paramètre `address` de la fonction est analysé. Ce paramètre peut contenir :

- Le nom d'un fichier : dans ce cas, le fichier est ouvert et le descripteur de ce fichier est renvoyé par la fonction.
- La chaîne de caractères "EMULATION" : un socket de la famille `AF_UNIX` et de type `SOCK_DGRAM` est créé. Les adresses nécessaires au socket sont stockées dans des structures prédéfinies et le descripteur de ce socket est renvoyé. Dans ce cas, tous les messages sont envoyés vers `PPutFlows`, qui est un autre processus, actif en arrière plan, où le protocole PRISM est implémenté au dessus de UDP.
- Une adresse IP qui se présente sous deux formes :
    1. Sous la forme d'un nom d'hôte : l'adresse IP numérique est obtenue par la fonction `gethostbyname()`. Si l'adresse IP numérique est valide, la fonction `OpenIP()` est appelée (création d'un socket), après stockage de l'adresse qui sera nécessaire lorsqu'un datagramme sera effectivement envoyé.
    2. Sous une forme numérique (chaîne de caractères en notation décimale) : la fonction `inet_addr()` est appelée, elle renvoie l'adresse sous la forme d'un entier long. Le reste de la procédure est identique au cas précédent.
- Une adresse PRISM : un socket de la famille `AF_PRISM`, de type `SOCK_DGRAM` est créé et un descripteur de socket est renvoyé.

4.5.5.2 POpenReceiver()

```
int POpenReceiver (  char       *address,
                     int        port )
```

La fonction `POpenReceiver()` est tout à fait similaire à `POpenSender()` et plus simple puisqu'aucune structure d'adresse n'est conservée pour usage ultérieur. En

conséquence, nous ne la décrirons pas ici. Dans le cas d'une connection internet, la fonction `OpenIpAndBind()` est utilisée en lieu et place de `OpenIp()` de façon à préciser sur quel port l'application attend un datagramme.

4.5.5.3 OpenIp()

```
static int OpenIP(int addr)
```

Si l'adresse internet passée comme paramètre est de classe A, B ou C, un socket de la famille `AF_INET` et de type `SOCK_DGRAM` est créé et son descripteur est retourné. Rappelons que le type de l'adresse détermine le nombre de bits réservé au codage du réseau et de l'hôte. Si l'adresse est de classe D (adresse multicast), après la création du socket, plusieurs options sont ajoutées à l'aide de la fonction `setsokopt()` de façon à permettre une diffusion dans un domaine multicast.

4.5.5.4 OpenIPAndBind

```
static int OpenIPAndBind(int addr, int port)
```

Après la création du socket par `OpenIP()`, celui-ci est associé à un port local par l'intermédiaire de la fonction `bind()`. Le descripteur de socket est ensuite retourné.

4.5.5.5 Evolution de l'API de distribution

L'API de distribution se présente actuellement sous la forme de fonctions C dont nous venons de décrire les principales. La définition de cette API en termes d'une classe d'objets de type distribution est à l'étude.

4.5.6 GESTION DE LA COMMUNICATION PAR LE WHITEBOARD

4.5.6.1 Lancement de l'application

Au démarrage de l'application, les paramètres de la ligne de commande peuvent contenir une adresse ainsi qu'un numéro de flot. Les détails des types d'adresse possibles ont été définis précédemment.

Si l'option "-master" est ajoutée, la fonction `wbAddMaster()` est appelée. Dans le cas où une adresse valide (nous préciserons ce terme ultérieurement) suit cette option, un socket est créé par la fonction `wbCreatePrismOutputSocket()` pour l'envoi des paquets d'information. La fonction `wbAddMaster()` crée par ailleurs l'ensemble des outils (dessin, curseur...) de l'interface utilisateur du poste maître. Dans le cas où l'option "-slave" est

ajoutée, un socket pour la réception des messages est créé par la fonction `wbCreatePrismInputSocket()`.

4.5.6.2 Les fonctions wbOut(), wbSingle(), wbOutDouble() et WbOutRaw()

En fonction du type de message que le poste maître doit envoyer, une de ces fonctions est utilisée. Tout paquet contient un en-tête composé des champs suivants :

- timestamp : utilisé à des fins statistiques,
- action : un code de commande,
- paramètre : accompagne le champ action.

Les paquets de ce type sont créés par la fonction `wbOut()`. Des paquets contenant des informations complémentaires sont créés par les autres :

- `WbOutSingle()` pour transmettre les coordonnées d'un point,
- `WbOutDouble()` pour transmettre les coordonnées d'une paire de points,
- `WbOutRaw()` pour transmettre les coordonnées d'un point ainsi qu'un ensemble de données et leur longueur.

Lorsque l'utilisateur exécute une action qui a une répercussion au niveau de la fenêtre principale, une des fonction précédentes est appelée. Après remplissage des structures de données correspondant à l'en-tête du paquet d'information ainsi qu'aux éventuelles structures secondaires, l'ensemble est recopié dans une mémoire tampon. Ces informations sont transmises sur le réseau par l'intermédiaire de la fonction `sendto()` présentée précédemment.

4.5.6.3 La fonction wbIn()

La fonction `void wbIn()` analyse le paquet de données entrant et réalise les affichages adéquats

La fonction `XtAppAddInput()` définie dans la libraire XToolkit de XWindow autorise un programme à obtenir ses informations en entrée à partir d'un fichier et, par extension, d'un socket. Elle permet de préciser la fonction qui sera appelée lorsqu'un datagramme sera disponible.

Les paramètres de cette fonction sont :

- le contexte de l'application,
- le descripteur de fichier (de socket),
- un masque précisant le type d'opération possible sur le fichier (lecture ...),
- un pointeur vers la fonction à appeler,

- les données client, données passées comme paramètre lors de l'appel de la fonction enregistrée.

Dans le cas du whiteboard, la fonction appelée est `wbIn()`. Cette fonction a pour objet de lire le contenu d'un paquet à l'aide de la fonction `recvfrom` (socket en mode déconnecté). Le message est ensuite interprété en termes d'action d'affichage à entreprendre. La première partie du paquet de données est un champ `action` qui détermine la fonction spécifique à appeler et la façon d'analyser le reste du paquet.

5. ESSAI DE MISE EN OEUVRE DE L'API[4] DE DISTRIBUTION : CONCEPTION D'UNE VERSION MS WINDOWS DU WHITEBOARD

5.1 INTRODUCTION

Pour tester et montrer comment la plate-forme PRISM peut s'adapter à l'hétérogénéité au niveau des systèmes d'exploitation, il est intéressant de tenter une implémentation du Whiteboard dans l'environnement MS Windows (WinWhiteboard).

L'idée initiale est d'implémenter une partie suffisante de l'API de distribution afin de pouvoir évaluer les possibilités de communication entre un poste en mode maître dans l'environnement UNIX et un poste en mode esclave dans l'environnement MS WINDOWS. L'intérêt premier de ceci est de mettre en évidence les modifications éventuelles à apporter à la définition de cette API de façon à rester le plus indépendant du système sous-jacent.

Au-dessus de cette API, la version initiale du WinWhiteboard a été conçue avec une architecture qui devrait permettre de construire aisément une application complète.

L'approche de conception et de développement est orientée objet, cette approche respecte particulièrement les objectifs de PRISM en favorisant la réutilisation et en masquant la complexité de certaines opérations de bas niveau, ce qui nous a permis de travailler dans l'environnement Borland C++ 4.0 avec le support de la librairie OWL[5] et du générateur de code APP EXPERT.

Par ailleurs, la conception des interfaces homme-machine MS WINDOWS et XWINDOW est sensiblement différente. Il nous semblait intéressant de mettre en évidence les facteurs à respecter pour conserver l'homogénéité de l'interface d'applications distribuées.

5.2 MISE EN OEUVRE DE L'API DE DISTRIBUTION

Deux objectifs ont été poursuivis lors de la conception de cette API :

- Masquer au maximum les particularités liées à l'utilisation de MS WINDOWS et, plus spécifiquement, celles liées à l'utilisation des Winsockets par rapport aux versions UNIX de l'interface socket.

---

[4] Application programming Interface.

[5] Object Window Library

- Définir L'API de distribution sous la forme d'une classe d'objets "Distribution", pour les raisons évoquées en introduction, mais principalement pour nous rapprocher des spécifications définies dans un rapport interne de l'ENST Bretagne.

5.2.1 LES WINDOWS SOCKETS

La spécification des Windows Sockets définit une interface de programmation réseau pour Microsoft Windows qui est basée sur la notion de "socket" telle qu'elle est définie par la Berkeley Software Distribution (BSD) de l'Université de Californie à Berkeley et telle que nous l'avons introduite au point 4.4.4. Elle ajoute un certain nombre de routines spécifiques de façon à pouvoir générer des messages Windows à partir d'événements liés aux communications réseau.

Ces spécifications ont pour objectif de fournir une API unique à laquelle les développeurs d'applications puissent se conformer en même temps que les vendeurs de logiciels réseau. Ces spécifications définissent la syntaxe des fonctions ainsi que leur sémantique.

Les méthodes de classe Distribution font appel à l'interface du protocole UDP/IP que toutes les implémentations des Windows Sockets sont sensées supporter.

5.2.2 CONCEPTION DE L'API DE DISTRIBUTION

Ce paragraphe reprend l'ensemble des méthodes publiques de la classe de distribution telle que nous l'avons implémentée. Le seul protocole mis en oeuvre dans cette première version est UDP/IP. Pour la description des méthodes privées de la classe, le lecteur peut consulter le code en annexe.

Des différences existent entre la spécification de l'API provenant du document interne de l'ENST Bretagne et l'implémentation définie ici, les méthodes `GetStatus()` et `DispatchReceive()` n'étant pas définies dans ce document.

5.2.2.1 Méthodes de la classe de distribution

`TDistribution (WINDOWREF aWindow)`

Le constructeur de la classe ne réalise aucune action particulière. Une copie du paramètre WINDOWREF est conservée, ce paramètre peut servir à déterminer la fenêtre susceptible de gérer la réception d'un paquet. Dans l'implémentation présente, celui-ci est défini comme étant de type HWND. Dans l'environnement X WINDOW, il pourrait être de type Widget.

`~TDistribtion ()`

Le destructeur réalise les opérations de libération de la mémoire propres à l'utilisation des Winsockets.

`int GetStatus()`

Cette méthode retourne l'état de l'objet.

`void SetFlow(int port)`

Cette méthode détermine le numéro de flot à partir d'un numéro de port passé comme paramètre. Remarquons que dans cette implémentation aucune vérification n'est faite sur la validité du numéro de port passé comme paramètre.

`int SetAddress( const char FAR * name )`

La première opération réalisée par la méthode[6] est l'analyse du format de la chaîne de caratères passée comme paramètre, le format identifie l'espace utilisé (voir Le bloc de distribution de PRISM dans le chapitre 2). Cette méthode réalise ensuite un ensemble d'opérations propres à l'utilisation des Winsockets telles que localiser, charger en mémoire et vérifier la version de la librairie dynamique "winsock.dll". C'est à ce moment que sont également déterminées les adresses des procédures de la librairie. Une variable d'état est mise à jour lors de ces vérifications, l'état de l'objet peut être testé par l'intermédiaire de la méthode `GetStatus()`. Si l'utilisateur de l'objet fait appel à `GetStatus()` juste après l'appel de `SetAddress()`, il pourra adopter un comportement adéquat.

Cette approche nous semble préférable à celle qui consiste à créer des méthodes de recherche et de vérification de la librairie dynamique publique car celles-ci ont une sémantique propre à l'environnement MS WINDOWS.

`SetAddress()` fixe l'adresse de réception ou d'émission. Dans cette implémentation, il s'agit d'une adressse IP numérique déterminée à partir d'une chaîne de caractères représentant un "hostname". Si ce dernier est valide, elle retourne 1 ou 0 dans le cas contraire.

`void OpenAsReceiver()`

Cette méthode donne à l'objet la propriété de "récepteur". Dans cette implémentation, elle crée un socket pour la réception. Seul le protocole UDP est supporté.

---

[6] Dans l'implémentation présente aucune analyse n'est effectuée puisque un seul espace est implémenté.

```
int   Receive(char *buf, int lg)
```

La méthode attend un paquet en entrée et le place dans le tampon passé comme premier paramètre et de longueur `lg`.

```
int   DispatchReceive()
```

Cette méthode met en place le mécanisme qui permet de confier à une fenêtre la gestion de la réception des paquets. Dans cette implémentation, la fenêtre recevra un message `WM_SOCKET` en cas d'arrivée d'un datagramme. Remarquons que la méthode conserve sa sémantique dans d'autres environnements comme XWindow où l'implémentation d'un mécanisme similaire est possible.

```
void  Close()
```

La méthode ferme l'objet en tant qu'émetteur ou récepteur.

5.2.2.2 Perspectives

Une version ultérieure devrait autoriser l'emploi d'autres protocoles de communication, de façon à augmenter la fiabilité des communications. Des recherches dans ce sens sont actuellement menées à l'ENST Bretagne. Dans ce cadre, un essai d'implémentation de PTP sur PC a été tenté.

Une étape intermédiaire serait d'implémenter le mode "émulation". Rappelons que le mode émulation consiste à implémenter le protocole PTP au-dessus de UDP. Une façon intéressante de réaliser cette implémentation serait de la réaliser sous la forme d'une librairie dynamique, ce qui permettrait de conserver la même interface de programmation et faciliterait les modifications ultérieures.

Une autre possibilité d'extension de l'application serait d'implémenter l'accès en entrée et en sortie sous la forme de fichiers (espace PRISM_FILE). C'est probablement l'extension qui est la plus simple à réaliser si on ne tient pas compte des problèmes de temporisations et de synchronisation.

5.3 CONCEPTION DU WINWHITEBOARD

5.3.1 SPÉCIFICATION DU WINWHITEBOARD

Dans sa version actuelle, les scénarios d'utilisation du Whiteboard (indépendamment de la plate-forme) ne sont pas encore clairement définis. Son interface utilisateur n'est donc pas encore définitivement arrêtée. En conséquence, les objets de l'application actuelle risquent

d'être utilisés dans un contexte différent de celui d'une éventuelle version ultérieure. Ce fait détermine l'architecture de l'application.

La version actuelle fonctionne uniquement en mode esclave mais une version ultérieure doit pouvoir fonctionner en mode maître. Les modes d'utilisation future du Whiteboard n'étant pas encore clairement définis, il paraît intéressant de concevoir l'application de façon à ce que les modes esclave et maître puissent être actifs simultanément.

5.3.2 ARCHITECTURE DE L'APPLICATION

Intuitivement, si l'on suit une approche orientée objet, et si l'on part des fonctionnalités principales de l'application, deux objets initiaux doivent être définis : l'objet "esclave" et l'objet "maître".

Ces deux objets ont seuls la responsabilité des échanges avec l'extérieur. Pour réaliser ces échanges, ils font directement appel aux services définis dans l'API de distribution. Si une version ultérieure était développée, il est probable qu'elle ne fasse qu'indirectement appel à cette API à travers les services du bloc de gestion lorsque ce bloc aura été finalisé.

Les objets "esclave" et "maître" ont également la responsabilité de faire appel aux objets d'interface homme-machine.

A un objet "esclave" ou "maître" peut être associé un ensemble d'objets graphiques affichés à l'écran, ceci suite à une interaction avec l'utilisateur ou la réception d'un message par l'objet de la classe distribution. En donnant à ces objets (`TRectangle`, `TCircle`, `TLine`...) une certaine persistance, la gestion des affichages et des rafraichissements de l'écran est facilitée. Par ailleurs, cette approche donne la possibilité de modifier des objets existants et déjà affichés mais cette fonctionnalité n'est pas actuellement implémentée.

La gestion de l'ensemble des objets graphiques d'un `TSlave` est de la responsabilité d'un objet de type `TDrawings`. Les classes d'objets graphiques actuellement définies sont : `TRectangle`, `TCircle` et `TLine`. Ces trois classes sont les sous-classes de `TDrawForms` dans laquelle une méthode virtuelle `Draw()` est définie, celle-ci est redéfinie par chaque classe d'objets graphiques. L'affichage de l'ensemble des objets d'un `TDrawings` est ainsi fortement simplifié.

Seul l'objet esclave (`TSlave`) est actuellement défini. Néanmoins, l'architecture proposée permettrait aisément, si l'évolution des besoins le nécessitait, de créer plusieurs objets `TSlave` ou `TMaster` totalement indépendants, chacun pouvant éventuellement être associé à une fenêtre différente

Ces quelques principes nous permettent de proposer une architecture (voir figure 5-1). Nous utiliserons la notation de G. Booch [4] pour la représentation des relations[7] entre classes d'objets.

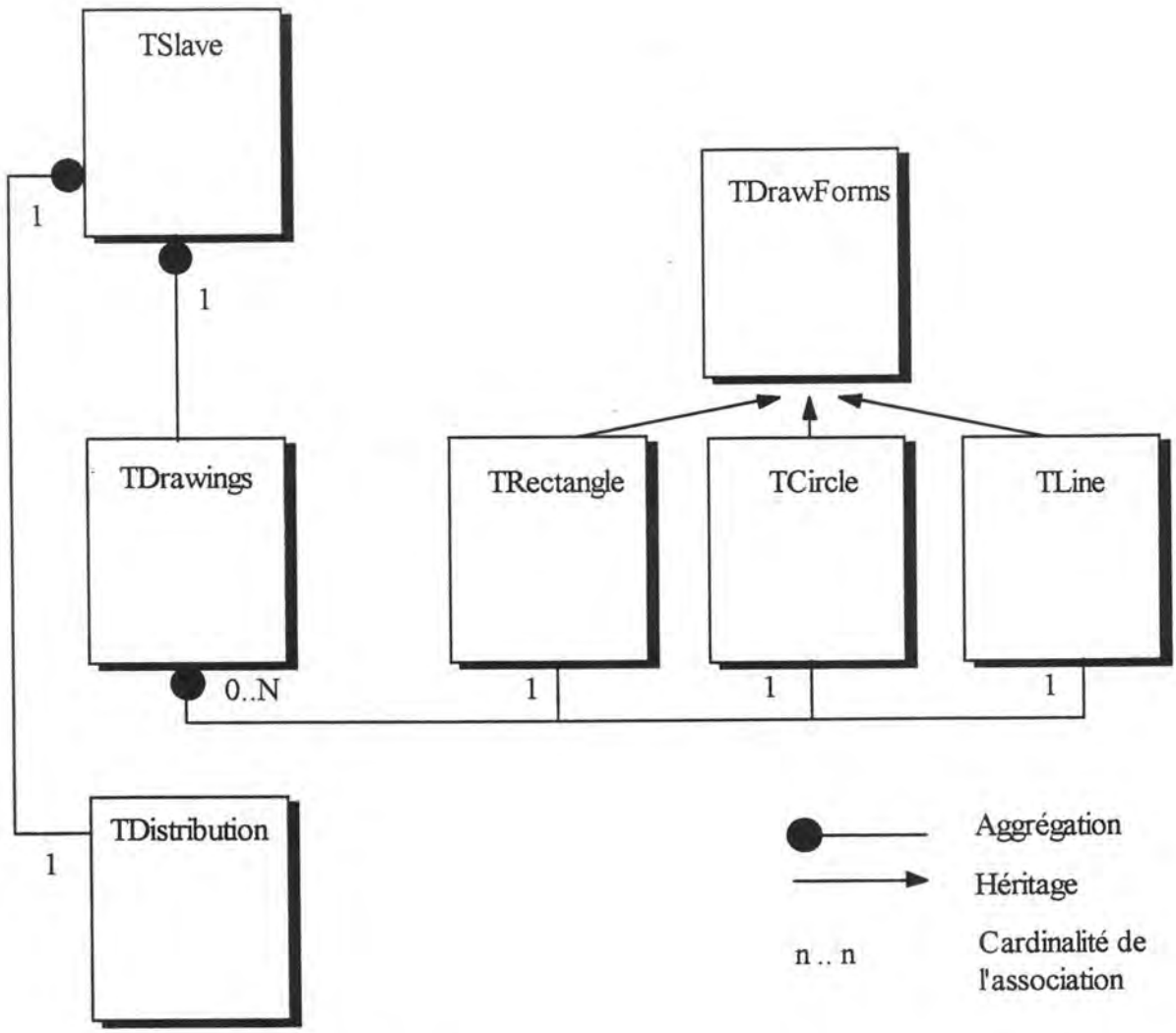


Figure 5–1 : Architecture du Win Whiteboard

5.3.3 PRINCIPALES CLASSES DE L'APPLICATION ET LEURS MÉTHODES

Nous reprenons ici les principales classes à l'exception de la classe Distribution qui a été décrite précédemment. Par ailleurs, nous renvoyons le lecteur aux annexes pour l'examen des classes spécifiques à l'environnement de programmation telles que la classe `winpwbApp`

---

[7] G. Booch [4] définit un objet agrégé de la façon suivante : *"An object composed of one or more other objects, each of which is considered a part of the aggregate object".*

dérivée de la classe `TApplication` de la librairie OWL. Dans la description, nous utiliserons le terme de "méthode" à la place de "fonction membre" spécifique au C++.

5.3.3.1 TSlave

```
TSlave(    TWindow *notifyWindow
           TWindow *slideWindow, char *address,
           char *port, char *slides)
```

Le constructeur de la classe `TSlave` recoit comme paramètre l'adresse de la fenêtre qui va gérer les événements réseau ainsi que l'adresse de la fenêtre d'affichage, les trois derniers paramètres sont nécessaires pour la création d'une instance d'un objet de type `TDistribution`.

```
~TSlave()
```

Le destructeur appelle la méthode `close` de l'objet `TDistribution` associé.

5.3.3.2 TDrawForms

```
TDrawForms(TColor *currentColor, TDC *aDC)
virtual void Draw()
```

Cette méthode virtuelle n'engendre aucune action ni changement d'état, sa seule fonction est de pouvoir la redéfinir par héritage.

5.3.3.3 TDrawings

```
TDrawings(TWindow *slideWindow)
void SetColors()
```

La méthode remplit une table de pointeurs vers des objets de type `TColor` (défini dans la librairie OWL). Ces couleurs sont les mêmes que celles définies dans la version XWindow du Whiteboard.

```
void GetCurrentColor(unsigned long newcolor)
```

La méthode met à jour la variable membre définissant la couleur courante à partir de l'index passé comme paramètre.

```
void DrawCircle (int X1, int X2, int Y1, int Y2)
```

La méthode crée un nouvel objet `TCircle` et insère son adresse dans un tableau de `TDrawForms`.

```
void DrawLine (int X1, int X2, int Y1, int Y2)
```

La méthode crée un nouvel objet TLine et insère son adresse dans un tableau de TDrawForms.

```
void DrawBox(int X1, int X2, int Y1, int Y2)
```

La méthode crée un nouvel objet TRectangle et insère son adresse dans un tableau de TDrawForms.

```
void DrawAll()
```

La méthode trace l'ensemble des TDrawForms associés.

```
void ClearAll()
```

La méthode détruit l'ensemble des TDrawForms associés.

5.3.3.4 TDrawRectangle

```
TDrawRectangle(      TDC *aDC,  TColor *currentColor,
                  TPoint *pt1, TPoint *pt2 )
```

Le créateur conserve dans des variables membres la couleur et les coordonnées du rectangle.

```
virtual void Draw()
```

La méthode trace le rectangle.

5.3.3.5 TDrawLine

(voir TDrawRectangle)

```
TDrawLine (    TDC *aDC, TColor *currentColor,
               TPoint *pt1 , TPoint *pt2 )
virtual void Draw()
```

5.3.3.6 TDrawCircle

(voir TDrawRectangle)

```
TDrawCircle (  TDC *aDC, TColor *currentColor,
               TPoint *pt1 , TPoint *pt2 )
virtual void Draw()
```

5.3.4 IMPLÉMENTATION DE LA RÉCEPTION DES PAQUETS

Les quelques notes qui suivent devraient aider le lecteur à comprendre le mécanisme central de l'application, à savoir l'attente de l'arrivée d'un message, l'examen de son contenu et l'exécution des instructions d'affichage adéquates.

La méthode `DispatchReceive()` de l'API de distribution est appelée par le créateur de toute instance d'un objet `TSlave`. Cette méthode fait appel à la fonction `WSAAsyncSelect` définie dans la spécification des Windows Sockets. La déclaration de cette fonction est reprise ci-dessous.

```
int PASCAL FAR (     SOCKET s,
                     HWND hWnd,
                     unsigned int wMsg,
                     long lEvent )
```

Après l'appel de cette fonction, lorsqu'un événement spécifié par le paramètre `lEvent` se produira au socket `s`, la fenêtre spécifiée par le paramètre `hWnd` recevra un message dont le type est défini par le paramètre `wMsg`. Le type de message que nous avons défini est identifié par la constante `WM_SOCKET`.

Dans notre cas, la fenêtre qui recoit le message est la fenêtre principale de l'application. Un traitement approprié à la réception de ce message est défini : il consiste à aller lire le message par l'intermédiaire de la fonction `wbIn()` dont le corps est très proche de son homologue XWindow.

Une différence importante est toutefois à signaler : lors de la réception d'un message contenant une demande d'affichage d'un objet graphique, celui-ci est simplement créé et ajouté à la liste maintenue par le `TDrawings` de l'esclave. Le contenu de la fenêtre d'affichage est alors explicitement invalidé ce qui induit l'appel de la méthode virtuelle `Paint`. Cette dernière enfin se contente d'appeler la méthode virtuelle `Draw()` de chaque objet graphique créé.

Bien que le mécanisme décrit ci-dessus soit opérationnel, la réception des datagrammes pose encore des problèmes, ceux-ci étant lié à l'utilisation d'un protocole de communication sans connection.

5.3.5 GESTION DE LA COULEUR

La gestion actuelle des couleurs par le Win Whiteboard est suffisante pour montrer comment une couleur peut être associée à un objet graphique de l'application (rectangle, cercle...). Elle est totalement insuffisante pour afficher correctement des images.

Les particularités de la gestion de la couleur par le Whiteboard a été abordée précédemment. Dans le cas de la version Windows, il nous semble possible d'arriver à un résultat similaire.

La palette du système est un tableau caractérisant un sous-ensemble des couleurs qui peuvent être affichées simultanément à l'écran. Elle est partagée par toutes les applications Windows. L'API de Windows offre un ensemble de fonctions qui peuvent modifier les couleurs de cette palette. Le code supportant ces fonctions fait partie d'une librairie dynamique appelée pilote d'écran. Un pilote d'écran convertit les fonctions graphiques en appels spécifiques au matériel. Malheureusement, tous les pilotes d'écran ne supportent pas les fonctions de gestion de la palette. Pour déterminer les possibilités offertes par un pilote déterminé, Windows définit la fonction `GetDeviceCaps()`.

De façon à limiter les problèmes liés à la manipulation directe de la palette active, Windows définit la notion de "palette logique" (Logical Palette). Cette palette est associée au contexte graphique de chaque application et peut aisément être modifiée. Un ensemble de messages est par ailleurs défini pour informer les applications que des changements ont été apportés à la palette du système.

Pour visualiser les changements apportés à la palette logique, celle-ci est substituée à la palette du système par l'appel de la fonction `RealizePalette()`. La solution au problème posé se trouve donc vraisemblablement dans l'utilisation des palettes logiques.

5.3.6 CONCEPTION DE L'INTERFACE

L'interface actuelle est construite sur base d'un scénario d'utilisation minimal, que l'architecture proposée permet d'étendre aisément.

L'utilisateur a devant lui la fenêtre de l'application (figure 5-2).

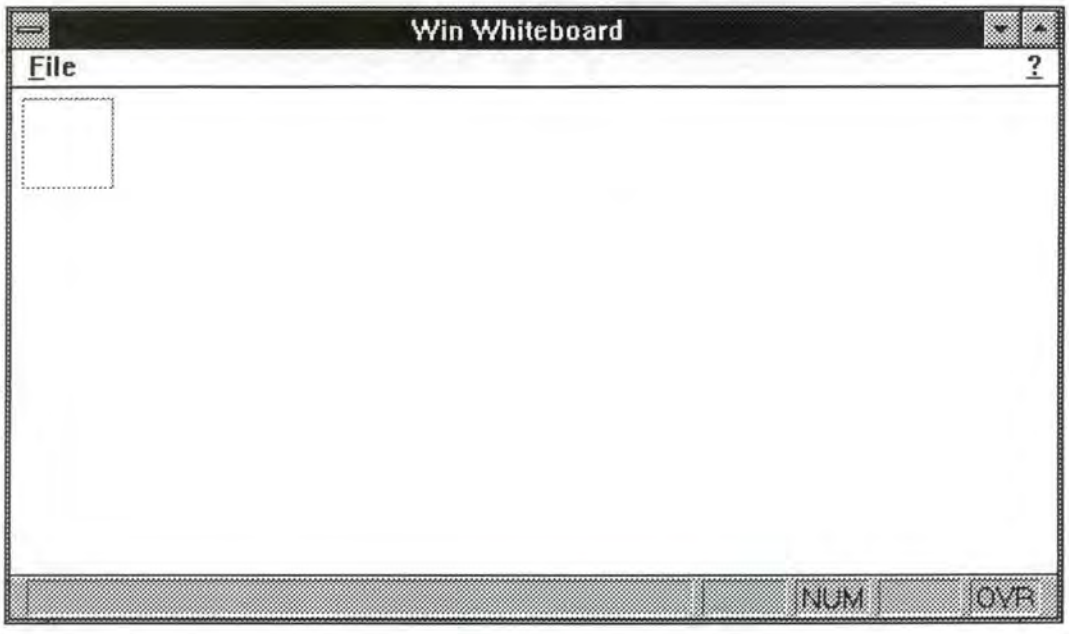


Figure 5–2 : Fenêtre principale du Win Whiteboard

Le menu "File" de cette fenêtre lui propose deux choix (seul le premier est implémenté) :

- Mode "esclave" : Après sélection de l'item du menu, une boite de dialogue (voir figure 5-3) lui est présentée permettant de spécifier une adresse et un numéro de port. A la fermeture de cette boite, l'objet "esclave" est créé et l'application est prête à recevoir des paquets. Un champ "Slides" a été ajouté de façon à pouvoir préciser dans une version ultérieure un nom de fichier, ce dernier contenant la référence des images utilisées pour une présentation. Remarquons que ce paramètre n'est ajouté ici que par souci d'homogénéité par rapport à la version initiale de l'application. Le champ d'édition pourrait par ailleurs être remplacé par un bouton dont le déclenchement serait associé à l'apparition d'une classique boite de dialogue de sélection de fichier.

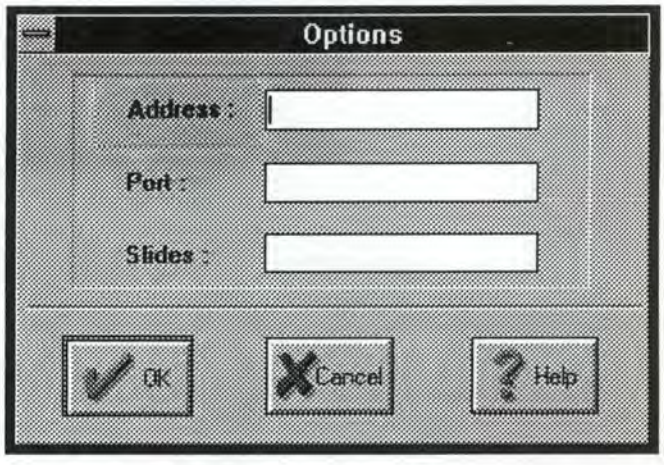


Figure 5–3 : Options de l'objet esclave

- Mode "maître" : Après sélection de l'item de menu, une boite de dialogue est présentée à l'utilisateur, permettant de spécifier une adresse et un numéro de port. A la fermeture de cette boite, l'objet "maître" est créé et l'application est prête à émettre. La création de

l'objet "maître" entraîne celle d'une barre d'outils supplémentaire. Les outils de manipulation des slides sont, eux, regoupés dans une boite de dialogue amodale.

Dans la version actuelle, tous les affichages se font dans la fenêtre principale de l'application et la création de plusieurs objets "esclaves" n'est pas autorisée. Dans des versions ultérieures, il faudrait associer une nouvelle fenêtre à chaque objet "maître" ou "esclave" créé, ce que l'architecture actuelle et l'environnement de programmation devraient permettre de réaliser très simplement. Ces fenêtres seraient des fenêtres filles de la fenêtre principale de l'application à laquelle elles resteraient toujours physiquement et visuellement attachées.

5.3.7 REMARQUE SUR L'HOMOGÉNÉITE DES INTERFACES UTILISATEUR

Dans une version ultérieure de la version XWindow du Whiteboard, il nous semble intéressant de regrouper tous les objets de l'interface du Whiteboard dans une seule et unique fenêtre de façon à éviter la dispersion des outils sur l'ensemble de l'écran. De plus, cette présentation est plus proche de ce qu'il est possible de réaliser sous Windows. Il pourrait être intéressant également de regrouper l'ensemble des outils du poste maître sous la forme d'une barre d'outils positionnée au sommet de la fenêtre principale.

6. CONCLUSIONS

De façon simplifiée, l'objectif de ce mémoire était de contribuer à l'adaptation à un environnement hétérogène de la plate-forme PRISM. L'essentiel du travail a porté sur la modification du Whiteboard, outil de haut niveau de gestion d'un espace de travail commun destiné à des applications multimédia.

Les modifications suivantes ont été introduites :

1. L'application, conçue initialement pour le système SUN OS, a été adaptée au système ULTRIX 3.4.
2. L'application peut maintenant être exécutée sur des postes de travail ne disposant que d'un écran noir et blanc. Cet aspect du travail comporte l'analyse des propriétés de l'écran, l'implémentation d'une fonction de dégradation d'image (dithering) et de nombreuses modifications annexes du code de l'application concernant la couleur des objets d'interface.
3. Un mécanisme de conversion de format d'image simple a été implémenté, il est basé sur l'utilisation de scripts exécutés lors de la phase de démarrage de l'application. Seule la conversion d'image au format PCX a été incluse mais la très grande simplicité des modifications à apporter dorénavant au code de l'application offre la possibilité d'étendre sensiblement le nombre de formats d'image acceptables par l'application.
4. Le prototype d'une version du Whiteboard adaptée à l'environnement Microsoft Windows a été développé. L'approche orientée objet et l'architecture proposée devraient permettre des modifications aisées de l'application en fonction de choix ultérieurs. La conception de ce prototype nous a permis de mettre en évidence les possibilités de concevoir une API de distribution très proche de ce qui existe dans l'environnement UNIX. Nous avons proposé dans cette partie du travail quelques possibilités d'extension.
5. Les particularités des interfaces utilisateurs ont été mises en évidence et nous avons proposé des solutions susceptibles de favoriser l'homogénéité de l'outil quelle que soit la plate-forme utilisée.

7. TABLE DES FIGURES

8. BIBLIOGRAPHIE

[1] R. BENTLEY, T. RODDEN, P. SAWYER, I. SOMMERVILLE, "Architectural Support for Cooperative Multiuser Interfaces", Computer, Vol. 27, N° 5, 1994, pp. 37-45.

[2] G. BOOCH, Object-oriented analysis and design, Benjamin/Cumming Publishing Company, Inc. California, 1994.

[3] D. E. COMER, Internetworking with TCP/IP, Vol. 1, principles, protocols and architecture, second edition, Prentice-Hall, Inc, Englewood Cliffs, 1991.

[4] C. RADU, "An Investigation of distributed multimedia domain : concept and applications", Research Paper 59/93, Institute of Computer Science, 1993.

[5] O. HUBER, "The WHITEBOARD: A Hight Level Multimedia Service for the PRISM platform", Diplomarbeit, 1994.

[6] T. LIANG et al., "When Client/Server Isn't Enought : Coordinating Multiple Distributed tasks", Computer, Vol. 27, N° 5, pp. 73-79.

[7] P. MOREL, A. TOMASSIO, " TCP_IP : Programmation Réseau : Les sockets", travail de fin d'étude, Ecole Nationale Supérieure de Mathématiques Appliquées et d'informatique de Grenoble ENSIMAG, 1993.

[8] A. NYE, The Definitive Guides to the X Window System, vol. 1, Xlib programming Manual, O'Reilly & Associates, Inc. United States of America, 1992.

[9] A. NYE, The Definitive Guides to the X Window System, vol. 4, X Toolkit Intrinsics programming manual, O'Reilly & Associates, Inc. United States of America, 1992.

[10] F. TOUTAIN et L. TOUTAIN," PTP : Un Protocole pour la diffusion de données multimédia", Rapport de l'Ecole nationale Supérieure des télécommunications de Bretagne.

[11] J. VANDERDONCKT, "Corpus ergonomique minimal des applications de gestion", Institut d'informatique, Projet Trident, FUNDP, Namur, 1992.

[12] M. WEBER, J. SCHEITZER, G. VÖLKSEN, "CSCW Tools : Concepts and Architecture", Computer, Vol. 27, N° 2, 1994, pp. 28-36.

[13] XloadImage, command xloadimage, inclus dans X-Window, Version 11, RRelease 5, X-Consortium.

[14] D. A. YOUNG, X Window Programmation avec les Xt Intrinsics, Masson Paris, Prentice Hall, London, 1991.

9. INDEX

U

V

W

X

FACULTES UNIVERSITAIRES N.D. DE LA PAIX, NAMUR
INSTITUT D'INFORMATIQUE

Année académique 1994-1995

CONTRIBUTION AU PROJET PRISM : AMELIORATION DU WHITEBOARD, UN OUTIL MULTIMEDIA DE HAUT NIVEAU

Michel Van Asten

ANNEXES

Mémoire présenté pour l'obtention du grade de Maître en Informatique

TABLE DES MATIERE

1. WIN WHITEBOARD

1.1 DISTRIB.H

```
#if !defined(__distrib_h)              // Sentry, use file only if it's
                                       // not already included.
#define __distrib_h

#include <winsock.h>
#include <windows.h>
#include <mem.h>
#include <string.h>

// define newMessage for socket entry
#define WM_SOCKET                      (WM_USER+100)
#define V11                            129
#define OPENDLL                        1
#define CLOSEDLL                       0
#define NOERROR                        1
#define DISTRIBUTION_ERROR                    0

// define for portability
#define WINDOWREF                      HWND


class TDistribution
{
public :
      TDistribution (WINDOWREF aWindow)
      {
            dllStatus = CLOSEDLL;
            status = NOERROR;
            notifyWindow = aWindow;
      }
      ~TDistribution()
      {
            ClearWinsocDLL();          // correct close of winsock.dll
      }
      void        SetFlow(int port);
      int         SetAddress( const char FAR * name );
      int         OpenIp();
      int         OpenIPAndBind();
      int         OpenAsReceiver();
      int         OpenAsSender();
      int         Receive(char *buf, int lg);
      int         DispatchReceive();
      void        Close();
      int         GetStatus(){return status;}
private:
      // correct load and free the winsoc.dll
      int         GetWinsocDLL(void);
      void        ClearWinsocDLL(void);
      void        DebugReceive(int nbytes);

      int         status;
      int         dllStatus;
      HINSTANCE   hinstWinSoc;


protected:
      sockaddr                         destination;
      int                              socID;
      int                              flow;
      struct hostent                   *resp;
      HWND                             notifyWindow;

      // handle to the module of dll
```

```
        // pointer to function to use winsock.dll
        int PASCAL FAR (FAR *fpWSAStartup)      (       WORD wVersionRequired,
                                                        LPWSADATA lpWSAData);
        int PASCAL FAR (FAR *fpWSACleanup)(void);
        int PASCAL FAR (FAR *fpWSAAsyncSelect) (        SOCKET s, HWND hWnd,
                                                        u_int wMsg,
                                                        long lEvent);
        SOCKET PASCAL FAR (FAR *fpsocket)       (       int af, int type,
                                                        int protocol);
        int PASCAL FAR (FAR *fpclosesocket)     (       SOCKET s);
        struct hostent FAR * PASCAL FAR         (       FAR *fpgethostbyname)(
                                                        const char *name);
        int PASCAL FAR (FAR *fprecvfrom)                (       SOCKET s, char *buf,
                                                        int len, int flags,
                                                        struct sockaddr *from,
                                                        int * fromlen);
        int PASCAL FAR (FAR *fpbind)            (       SOCKET s,
                                                        const struct sockaddr *addr,
                                                        int namelen);
);

#endif
```

1.1.1 DISTRIB.CPP

```
#include <windows.h>
#include "distrib.h"


int TDistribution::GetWinsocDLL()
{
        WORD            wVersionRequested;
        WSADATA         wsaData;
        char            sysPath[500];
        int             err;


        wVersionRequested = V11;
        GetSystemDirectory(sysPath, 500);       // get the path of the system
        strcat(sysPath, "\\");
        strcat(sysPath, "winsock.dll");

        hinstWinSoc = LoadLibrary(sysPath);
        // Getting adress of dll functions
        if (hinstWinSoc > 32) {
                (FARPROC) fpbind = GetProcAddress( hinstWinSoc, "bind" );
                (FARPROC) fpWSAStartup = GetProcAddress(hinstWinSoc, "WSAStartup");
                (FARPROC) fpWSACleanup = GetProcAddress(hinstWinSoc, "WSACleanup");
                (FARPROC) fpWSAAsyncSelect = GetProcAddress( hinstWinSoc,

        "WSAAsyncSelect");
                (FARPROC) fpsocket = GetProcAddress(hinstWinSoc, "socket");
                (FARPROC)       fpgethostbyname      =      GetProcAddress(hinstWinSoc,
"gethostbyname");
                (FARPROC) fprecvfrom = GetProcAddress(hinstWinSoc, "recvfrom");
        }
        else {
                dllStatus = CLOSEDLL;
                return status=DISTRIBUTION_ERROR;
        }

        err = (*fpWSAStartup)( wVersionRequested, &wsaData );
        if ( err != 0 ) {
                // Tell the user that we couldn't find a useable
                // winsock.dll.
                 dllStatus = OPENDLL;
                 return status=DISTRIBUTION_ERROR;
        }

        if ( LOBYTE( wsaData.wVersion ) != 1 ||
                        HIBYTE( wsaData.wVersion ) != 1 ) {
                // Tell the user that we couldn't find a useable
                // winsock.dll.
```

```
            dllStatus = OPENDLL;
            ClearWinsocDLL();

            return status=DISTRIBUTION_ERROR;
      }
      return NOERROR;

}

void TDistribution::ClearWinsocDLL()
{
      if (dllStatus == OPENDLL) {
            (*fpWSACleanup) ();
            FreeLibrary(hinstWinSoc); // winsock.dll will not be futher used
            dllStatus = CLOSEDLL;
      }
}

void TDistribution::Close()
{
      (*fpclosesocket) ( socID );
}


void TDistribution::SetFlow(int port)
{
      // no control done yet !!
      flow = port;
}

int TDistribution::SetAddress( const char FAR * name )
{
      // the Winsoc.dll is loaded here because in further implementation
      // the space will not be necessary PRISM_INTERNET so the string
      // name would be check here before loading the dll.
      status = GetWinsocDLL();

      if (  status == NOERROR )
            if ( resp =  (*fpgethostbyname) ( name ) )
                  return NOERROR;

      return status=DISTRIBUTION_ERROR;
}

int TDistribution::OpenIp()
{
      if(socID = (*fpsocket)(AF_INET, SOCK_DGRAM, 0))
            return NOERROR;
      else
            return status=DISTRIBUTION_ERROR;
}


int TDistribution::OpenIPAndBind()
{
      struct sockaddr_in dest;

      if (OpenIp()){
            memset(&dest, 0, sizeof(sockaddr_in)); // Set All structure to O
            dest.sin_family = AF_INET;
            dest.sin_port = flow;
            if((*fpbind) ( socID, (sockaddr*)&dest, sizeof(sockaddr_in) )==0)
                  return NOERROR;
      }
      return status=DISTRIBUTION_ERROR;
}

int TDistribution::OpenAsReceiver()
{
      if(OpenIPAndBind())
                  return NOERROR;
      return status=DISTRIBUTION_ERROR;
}

int TDistribution::OpenAsSender()
```

```
{
        return status=DISTRIBUTION_ERROR;
        // not yet implemented
}

int TDistribution::DispatchReceive()
{
        // if any FD_READ event occur at socID socket the notify Window
        // receaves a WM_SOCKET message
        return (*fpWSAAsyncSelect) ( socID, notifyWindow, WM_SOCKET, FD_READ );
}


int TDistribution::Receive(char *buf, int lg)
{
        int    nbytes;

        nbytes=(*fprecvfrom) ( socID, buf, lg, 0, NULL, 0);
        DebugReceive(nbytes);
        return nbytes;
}


void TDistribution::DebugReceive(int nbytes)
{
        switch(nbytes){
                case WSAENETDOWN:
                        MessageBox( NULL,
                                    "The Windows Sockets implementation has detected
                                     that the network subsystem has failed.",
                                    "Network error", MB_OK);
                break;

                case WSAEFAULT:
                        MessageBox( NULL,
                                    "The fromlen argument was invalid: the from buffer
                                    was too small to accommodate the peer address.",
                                    "Network error", MB_OK);
                break;

                case WSAEINTR:
                        MessageBox( NULL,
                                    "The (blocking) call was canceled via
                                    WSACancelBlockingCall()",
                                    "Network error", MB_OK);
                break;

                case WSAEINPROGRESS:
                        MessageBox( NULL,
                                    "A blocking Windows Sockets operation is in
                                    progress.",
                                    "Network error", MB_OK);
                break;


                case WSAEINVAL:
                        MessageBox( NULL,
                                    "The socket has not been bound with bind().",
                                    "Network error", MB_OK);
                break;

                case WSAENOTCONN:
                        MessageBox( NULL,
                                    "The socket is not connected (SOCK_STREAM only).",
                                    "Network error", MB_OK);
                break;

                case WSAENOTSOCK:
                        MessageBox( NULL, "The descriptor is not a socket.",
                                    "Network error", MB_OK);
                break;

                case WSAEOPNOTSUPP:
                        MessageBox( NULL,
```

```
                                "MSG_OOB was specified, but the socket is not of
                                type SOCK_STREAM.",
                                "Network error", MB_OK);
                break;

                case WSAESHUTDOWN:
                        MessageBox( NULL,
                                "The socket has been shutdown; it is not possible
to                              recvfrom() on a socket after shutdown() has been
                                invoked with how set to 0 or 2.",
                                "Network error", MB_OK);
                break;

                case WSAEWOULDBLOCK:
                        MessageBox( NULL,
                                "The socket is marked as non-blocking and the
                                        recvfrom() operation would block.",
                                "Network error", MB_OK);
                break;

                case WSAEMSGSIZE:
                        MessageBox( NULL,
                                "The datagram was too large to fit into the
                                specified buffer and was truncated.",
                                "Network error", MB_OK);
                break;

                case WSAECONNABORTED:
                        MessageBox( NULL,
                                "The virtual circuit was aborted due to timeout or
                                other failure.",
                                "Network error", MB_OK);
                break;
        }
}
```

1.1.2 TSLAVE.H

```
#if !defined(__tslave_h)
#define __tslave_h

#include <owl\owlpch.h>
#pragma hdrstop

#include "wnpwbapp.rh"
#include "distrib.h"
#include "wbPacket.h"
#include "tdraws.h"
#include <owl\statusba.h>

class TSlave{
public:
// methods
        TSlave(TWindow  *notifyWindow,  TWindow  *drawWindow,  char  *address,  char
*port, char *slides);
        ~TSlave(){if (source) source->Close();}
        void wbIn();
        int    GetStatus(){return status;}

// variables
        TDrawings               *pSlaveDrawings;
        TDistribution           *source;
        char                            *slidesFName;
private:
        int                             status;
};

#endif
```

1.1.3 TSLAVE.CPP

```
#include "tslave.h"

TSlave::TSlave(TWindow *notifyWindow, TWindow *drawWindow, char *address, char
*port, char *slides)
{
            status = NOERROR;
            pSlaveDrawings = new TDrawings(drawWindow);
            pSlaveDrawings->SetColors();
            source = new TDistribution(notifyWindow->HWindow);
            if(source->GetStatus()){
                  source->SetAddress(address);
                  source->SetFlow(atoi(port));
                  source->OpenAsReceiver();
                  source->DispatchReceive();
            }
            else{
                  status = DISTRIBUTION_ERROR;
                  delete(source);
                  source = NULL;
                  return;
            }
            // not yet implemented
            slidesFName = slides;
}

void TSlave::wbIn()
{
static      struct wbpacketheader     incomingpacketheader;
static      struct wbsubsinglecoord   incomingsubsingle;
static      struct wbsubdoublecoord   incomingsubdouble;
static      struct wbsubraw           incomingsubraw;
static      char                      text[80];
static      char                      buff[MAXPACKETDATA+16];
int                                   nbytes;
int                                   i;

if (source->Receive(buff, nbytes )> 0){


      memcpy( &incomingpacketheader, &buff[0], sizeof (struct wbpacketheader));

      // Get the DC for drawing
      // Decode the command
      switch (incomingpacketheader.action)
      {
      case WBDRAWCIRCLE:
            //      change drawing color if neccessary
            if      (incomingpacketheader.parameter      !=      pSlaveDrawings-
>currentIndexColor)
                  pSlaveDrawings-
>GetCurrentColor(incomingpacketheader.parameter);

            memcpy   (   &incomingsubdouble,         &buff[sizeof         (struct
wbpacketheader)],
                        sizeof (struct wbsubdoublecoord));
            pSlaveDrawings->DrawCircle (    incomingsubdouble.x1,
                                            incomingsubdouble.x2,
                                            incomingsubdouble.y1,
                                            incomingsubdouble.y2);
            break;
      case WBDRAWLINE:
            /*      change drawing color if neccessary */
            if      (incomingpacketheader.parameter      !=      pSlaveDrawings-
>currentIndexColor)
                  pSlaveDrawings-
>GetCurrentColor(incomingpacketheader.parameter);

            memcpy   (   &incomingsubdouble,         &buff[sizeof         (struct
wbpacketheader)],
                        sizeof (struct wbsubdoublecoord));
            pSlaveDrawings->DrawLine (            incomingsubdouble.x1,
                                                  incomingsubdouble.x2,
                                          incomingsubdouble.y1,
```

```
                                                incomingsubdouble.y2);
            break;
      case WBDRAWBOX:
            /*      change drawing color if neccessary */
            if      (incomingpacketheader.parameter      !=      pSlaveDrawings-
>currentIndexColor)
                  pSlaveDrawings-
>GetCurrentColor(incomingpacketheader.parameter);

            memcpy   (   &incomingsubdouble,         &buff[sizeof         (struct
wbpacketheader)],
                        sizeof (struct wbsubdoublecoord));

            pSlaveDrawings->DrawBox(  incomingsubdouble.x1,
                                      incomingsubdouble.x2,
                                      incomingsubdouble.y1,
                                      incomingsubdouble.y2);
            break;


      case WBCLEARWINDOW:
            pSlaveDrawings->ClearAll();
            break;

      default:
            return;
            break;
      }// switch

      }// if
}// wbIn()
```

1.1.4 TDRAWS.H

```
#if !defined(__tdraws_h)
#define __tdraws_h

#include <owl\owlpch.h>
#pragma hdrstop

#include <owl\dc.h>
#include <owl\color.h>
#include <owl\gdiobjec.h>

#define      MAXDRAWINGS 200


class TDrawForms{
public:
      TDrawForms(TColor *currentColor, TDC *aDC)
      {
            // Create a new TBrush owned by this object.
            rTBrush = new TBrush(*currentColor);
            pDC = aDC;
      }
      // The Draw method does nothing, the definition is only done for
      // inheritance
      virtual void Draw(){}
protected:
      TBrush            *rTBrush;
      TDC                     *pDC;
};

// $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
// $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$


      .
class TDrawings {
public:
      TDrawings(TWindow *aWindow)
      {
                              locDC = NULL;
```

```
                              listPosition=0;
                              DrawWindow = aWindow;
                              drawingList=       (TDrawForms**)
                                                 calloc(200,sizeof(void*));
                              currentIndexColor = 0;
      }
      void TDrawings::SetColors();
      void GetCurrentColor(unsigned long     newcolor);
      void SetDC(TDC *aDC)
      {
            locDC = aDC;
      }

      void DrawCircle (int X1, int X2, int Y1, int Y2);
      void DrawLine (int X1, int X2, int Y1, int Y2);
      void DrawBox(int X1, int X2, int Y1, int Y2);
      void DrawAll();
      void ClearAll();
public:
      TColor            *currentColor;     // a pointer to the current TColor
      TColor            *palette[256];     // a array of TColor, the colors
should                                       //    be    the    same    when
applications                                                 //    communicates
with others
      TDrawForms        **drawingList;     // a array of pointers to all the
                                           // drawings created by the the
                                           // TDrawings
      int               listPosition;           // the current position in the
list
      TDC               *locDC;            // the current Display context
      TWindow           *DrawWindow;       // the window to draw into
      unsigned long     currentIndexColor; // the index of the current Color
};



// $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
// $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$

class TDrawRectangle : public TDrawForms{
public:
      TDrawRectangle(   TDC *aDC,
                                          TColor *currentColor,
                                          TPoint *pt1, TPoint *pt2 )
                                          : TDrawForms(currentColor, aDC)
      {
                                     topLeftPt = pt1;
                                     bottomRight = pt2;

      }
      virtual void Draw(){
                                    TRect tempRect(     *topLeftPt,
                                                *bottomRight);
                                    // Use its own BRUSH for drawing.
                                    // We have only one instance of the TDC,
                                    // but as many BRUSH as TDrawForms
                                    pDC->SelectObject(*rTBrush);
                                    pDC->FrameRect( tempRect, *rTBrush);
      }
private:
      TPoint *topLeftPt;
      TPoint *bottomRight;
};

// $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
// $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$

class TDrawLine : public TDrawForms{
public:
      TDrawLine (     TDC *aDC,
                                    TColor *currentColor,
                                    TPoint *pt1 , TPoint *pt2 )
                                    : TDrawForms(currentColor, aDC)
      {
                                     beginPt = pt1;
```

```
                                        endPt = pt2;
        }
        virtual void Draw()
        {
                                        pDC->SelectObject(*rTBrush);
                                        pDC->MoveTo(*beginPt);
                                        pDC->LineTo(*endPt);
        }
private:
        TPoint *beginPt;
        TPoint        *endPt;
};


// $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
// $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$

class TDrawCircle : public TDrawForms{
public:
        TDrawCircle (       TDC *aDC, TColor *currentColor, TPoint *pt1 ,
                            TPoint *pt2 ): TDrawForms(currentColor, aDC)
        {
                                        topLeftPt = pt1;
                                        bottomRightPt = pt2;

        }
        virtual void Draw()
        {
                                        pDC->SelectObject(*rTBrush);
                                        pDC->Ellipse( *topLeftPt,
                                                        *bottomRightPt);
        }
private:
        TPoint *topLeftPt;
        TPoint *bottomRightPt;
};


#endif
```

1.1.5 TDRAWS.CPP

```
#include <owl\owlpch.h>
#pragma hdrstop

#include "tdraws.h"

void TDrawings::SetColors()
{
        palette[0] = new TColor (255,255,255);
        palette[1] = new TColor (0,0,0);
        palette[2] = new TColor (255,0,0);
        palette[3] = new TColor (0,255,0);
        palette[4] = new TColor (0,0,255);
        palette[5] = new TColor (0,255,255);
        palette[6] = new TColor (255,0,255);
        palette[7] = new TColor (255,255,0);
        palette[8] = new TColor (255*2/3,255*2/3,255*2/3);
        palette[9] = new TColor (255/3,255/3,25/3);
}


// Set the current Color index and the currentColor pointer
void TDrawings::GetCurrentColor(unsigned long       newcolor)
{
        if (newcolor < 0 || newcolor > 9)
                newcolor = 1;
        currentColor = palette[newcolor];
        currentIndexColor = newcolor;
}

// Call the Draw method of each object in the drawingList
```

```
void TDrawings::DrawAll()
{
      int   i=0;

      for(i=0; i<listPosition; i++){
            if(drawingList[i] != NULL)
                  drawingList[i]->Draw();
      }
}

// Create a new TDrawCircle and add it at the end of the drawingList
// Draw it with the TDrawCircle method
// Force the DrawWindow to refresh itself
void TDrawings::DrawCircle (int X1, int X2, int Y1, int Y2)
{
      if( listPosition < MAXDRAWINGS ){
            drawingList[listPosition]=
            new TDrawCircle (  locDC, currentColor, new TPoint(X1, Y1),
                               new TPoint(X2, Y2) );
            listPosition++;
            DrawWindow->Invalidate(TRUE);
      }
}

// Create a new TDrawLine and add it at the end of the drawingList
// Draw it with the TDrawLine method
void TDrawings::DrawLine (int X1, int X2, int Y1, int Y2)
{
      if( listPosition < MAXDRAWINGS ){
            drawingList[listPosition] =
            new TDrawLine (    locDC, currentColor, new TPoint(X1, Y1),
                               new TPoint(X2, Y2) );
            listPosition++;
            DrawWindow->Invalidate(TRUE);
      }
}

void TDrawings::DrawBox(int X1, int X2, int Y1, int Y2)
{
      listPosition++;
      if( listPosition < MAXDRAWINGS ){
            drawingList[listPosition]=
            new TDrawRectangle (locDC, currentColor, new TPoint(X1, Y1),
                               new TPoint(X2, Y2) );
            listPosition++;
            DrawWindow->Invalidate(TRUE);
      }
}

// Invoke the destructor of each TDrawForms in the drawingList.
// Force the DrawWindow to refresh itself.
void TDrawings::ClearAll()
{
      int i;

      for(i=0; i<listPosition; i++){
            delete drawingList[i];
      }
      listPosition = 0;
      DrawWindow->Invalidate(TRUE);
}
```

1.1.6 TSLAVEWN.H

```
#if !defined(__tslavewn_h)              // Sentry, use file only if it's
                                              // not already included.
#define __tslavewn_h

/*  Project winpwb

    Copyright © 1994. All Rights Reserved.

    SUBSYSTEM:      winpwb.apx Application
    FILE:           tslavewn.h
```

```
    AUTHOR:         MICHEL VAN ASTEN FUNDP 1995


    OVERVIEW
    ========
    Class definition for TSlaveWin (TWindow).
*/

#include <owl\owlpch.h>
#pragma hdrstop

#include <owl\window.h>

#include "wnpwbapp.rh"            // Definition of all resources.
#include "wnpwbapp.h"


//{{TWindow = TSlaveWin}}
class TSlaveWin : public TWindow {
public:
    TSlaveWin (     TWindow* parent, const char far* title = 0,
                    TModule* module = 0);
    virtual ~TSlaveWin ();

//{{TSlaveWinVIRTUAL_BEGIN}}
public:
    virtual void Paint (TDC& dc, BOOL erase, TRect& rect);
//{{TSlaveWinVIRTUAL_END}}

//{{TSlaveWinRSP_TBL_BEGIN}}
protected:
//{{TSlaveWinRSP_TBL_END}}
DECLARE_RESPONSE_TABLE(TSlaveWin);
};    //{{TSlaveWin}}


#endif                                          // __tslavewn_h sentry.
```

1.1.7 TSLAVEWN.CPP

```
/*  Project winpwb

    Copyright © 1994. All Rights Reserved.

    SUBSYSTEM:      winpwb.apx Application
    FILE:           tslavewn.cpp
    AUTHOR:         MICHEL VAN ASTEN FUNDP 1995


    OVERVIEW
    ========
    Source file for implementation of TSlaveWin (TWindow).
*/

#include <owl\owlpch.h>
#pragma hdrstop

#include "tslavewn.h"


//
// Build a response table for all messages/commands handled
// by the application.
//
DEFINE_RESPONSE_TABLE1(TSlaveWin, TWindow)
//{{TSlaveWinRSP_TBL_BEGIN}}
//{{TSlaveWinRSP_TBL_END}}
END_RESPONSE_TABLE;


//{{TSlaveWin Implementation}}
```

```
TSlaveWin::TSlaveWin (TWindow* parent, const char far* title, TModule* module):
    TWindow(parent, title, module){}


TSlaveWin::~TSlaveWin ()
{
    Destroy();
}


void TSlaveWin::Paint (TDC& dc, BOOL erase, TRect& rect)
{
    TWindow::Paint(dc, erase, rect);

    // INSERT>> Your code here.
    // for each TDrawForms the Draw method is call
        if (((winpwbApp*)GetApplication())->appSlave){
              ((winpwbApp*)GetApplication())->appSlave->pSlaveDrawings->SetDC(&dc);
              ((winpwbApp*)GetApplication())->appSlave->pSlaveDrawings->DrawAll();

        }
}
```

1.1.8 MODE.H

```
#if !defined(__mode_h)              // Sentry, use file only if it's not already
included.
#define __mode_h

/*  Project winpwb

    Copyright _ 1994. All Rights Reserved.

    SUBSYSTEM:    winpwb.apx Application
    FILE:         mode.h
    AUTHOR:       MICHEL VAN ASTEN FUNDP 1995


    OVERVIEW
    ========
    Class definition for Mode (TDialog).
*/

#include <owl\owlpch.h>
#pragma hdrstop

#include <owl\radiobut.h>
#include <owl\edit.h>
#include <owl\dialog.h>
#include <owl\validate.h>
#include <owl\module.h>

#include "wnpwbapp.rh"            // Definition of all resources.


//{{TDialog = Mode}}
//struct ModeXfer {
//{{ModeXFER_DATA}}
//    char    pTeditMode[ 255 ];
//    char    pTeditSlide[ 255 ];
//{{ModeXFER_DATA_END}}
//};


class Mode : public TDialog {
public:
       Mode (      char *Address, char *Port, char *Slides, TWindow* parent,
                   TResId resId = IDD_MODE, TModule* module = 0);
       virtual ~Mode ();
// variables
      char *locAddress, *locSlides, *locPort;
```

```
//{{ModeXFER_DEF}}
protected:
      TEdit *pTeditMode;
      TEdit *pTeditPort;
      TEdit *pTeditSlide;

//{{ModeXFER_DEF_END}}

//{{ModeRSP_TBL_BEGIN}}
protected:
       void BNClickedOK ();
       void BNClickedHelp ();
//{{ModeRSP_TBL_END}}
DECLARE_RESPONSE_TABLE(Mode);
};    //{{Mode}}


#endif                                          // __mode_h sentry.
```

1.1.9 MODE.CPP

```
/*  Project winpwb

    Copyright _ 1994. All Rights Reserved.

    SUBSYSTEM:    winpwb.apx Application
    FILE:         mode.cpp
    AUTHOR:       MICHEL VAN ASTEN FUNDP 1995


    OVERVIEW
    ========
    Source file for implementation of Mode (TDialog).
*/

#include <owl\owlpch.h>
#pragma hdrstop

#include "mode.h"


//
// Build a response table for all messages/commands handled
// by the application.
//
DEFINE_RESPONSE_TABLE1(Mode, TDialog)
//{{ModeRSP_TBL_BEGIN}}
       EV_BN_CLICKED(IDOK, BNClickedOK),
       EV_BN_CLICKED(IDHELP, BNClickedHelp),
//{{ModeRSP_TBL_END}}
END_RESPONSE_TABLE;


//{{Mode Implementation}}


Mode::Mode (char *Address, char *Port, char *Slides, TWindow* parent, TResId
resId, TModule* module):
    TDialog(parent, resId, module)
{
      pTeditMode = new TEdit(this, IDC_ADDRESS, 255);
      pTeditPort = new TEdit(this, IDC_PORT, 255);
      pTeditPort->SetValidator(new TRangeValidator(12345, 99999));
      pTeditSlide = new TEdit(this, IDC_SLIDELIST, 255);


      //SetTransferBuffer(&ModeData);


      // INSERT>> Your constructor code here.
      locAddress = Address;
      locPort = Port;
      locSlides = Slides;
```

```
}

Mode::~Mode ()
{
    Destroy();

    // INSERT>> Your destructor code here.

}


void Mode::BNClickedOK ()
{
      // INSERT>> Your code here.
      pTeditSlide->GetText(locSlides, 255);
      pTeditMode->GetText(locAddress, 255);
      pTeditPort->GetText(locPort, 255);
      CmOk();
}

void Mode::BNClickedHelp ()
{
      char  buf[500], mess[500];

      // load and merge separated ressources for help messages.
      GetModule()->LoadString(IDS_HELPADDRESS, mess, 500);
      GetModule()->LoadString(IDS_HELPFLOW, buf, 500);
      strcat(mess, buf);
      GetModule()->LoadString(IDS_HELPSLIDES, buf, 500);
      strcat(mess, buf);

      MessageBox(mess,"Receive\\Sending options", MB_OK);
}
```

1.1.10 WNPWBABD.H

```
#if !defined(__wnpwbabd_h)                    // Sentry, use file only if it's not
already included.
#define __wnpwbabd_h

/*  Project winpwb

    Copyright _ 1994. All Rights Reserved.

    SUBSYSTEM:    winpwb.exe Application
    FILE:         wnpwbabd.h
    AUTHOR:       MICHEL VAN ASTEN FUNDP 1995


    OVERVIEW
    ========
    Class definition for winpwbAboutDlg (TDialog).
*/


#include <owl\owlpch.h>
#pragma hdrstop

#include <owl\static.h>
#include "wnpwbapp.rh"                    // Definition of all resources.


//{{TDialog = winpwbAboutDlg}}
struct winpwbAboutDlgXfer {
//{{winpwbAboutDlgXFER_DATA}}
    char    autorCtrl[ 255 ];
//{{winpwbAboutDlgXFER_DATA_END}}
};


class winpwbAboutDlg : public TDialog {
public:
```

```
    winpwbAboutDlg (TWindow *parent, TResId resId = IDD_ABOUT, TModule *module =
0);
    virtual ~winpwbAboutDlg ();

//{{winpwbAboutDlgVIRTUAL_BEGIN}}
public:
    void SetupWindow ();
//{{winpwbAboutDlgVIRTUAL_END}}

//{{winpwbAboutDlgXFER_DEF}}
protected:
    TStatic *autorCtrl;

//{{winpwbAboutDlgXFER_DEF_END}}
};    //{{winpwbAboutDlg}}


// Reading the VERSIONINFO resource.
class ProjectRCVersion {
public:
    ProjectRCVersion (TModule *module);
    virtual ~ProjectRCVersion ();

    BOOL GetProductName (LPSTR &prodName);
    BOOL GetProductVersion (LPSTR &prodVersion);
    BOOL GetCopyright (LPSTR &copyright);
    BOOL GetDebug (LPSTR &debug);

protected:
    LPBYTE      TransBlock;
    void FAR    *FVData;

private:
    // Don't allow this object to be copied.
    ProjectRCVersion (const ProjectRCVersion &);
    ProjectRCVersion & operator =(const ProjectRCVersion &);
};


#endif                                          // __wnpwbabd_h sentry.
```

1.1.11 WNPWBABD.CPP

```
/*  Project winpwb

    Copyright _ 1994. All Rights Reserved.

    SUBSYSTEM:    winpwb.exe Application
    FILE:         wnpwbabd.cpp
    AUTHOR:       MICHEL VAN ASTEN FUNDP 1995


    OVERVIEW
    ========
    Source file for implementation of winpwbAboutDlg (TDialog).
*/


#include <owl\owlpch.h>
#pragma hdrstop

#include <owl\static.h>

#if !defined(__FLAT__)
#include <ver.h>
#endif

#include "wnpwbapp.h"
#include "wnpwbabd.h"


ProjectRCVersion::ProjectRCVersion (TModule *module)
{
    char    appFName[255];
```

```
    char    subBlockName[255];
    DWORD   fvHandle;
    UINT    vSize;

    FVData = 0;

    module->GetModuleFileName(appFName, sizeof(appFName));
    DWORD dwSize = GetFileVersionInfoSize(appFName, &fvHandle);
    if (dwSize) {
        FVData  = (void FAR *)new char[(UINT)dwSize];
        if (GetFileVersionInfo(appFName, fvHandle, dwSize, FVData)) {
            // Copy string to buffer so if the -dc compiler switch (Put constant
strings in code segments)
            // is on VerQueryValue will work under Win16.  This works around a
problem in Microsoft's ver.dll
            // which writes to the string pointed to by subBlockName.
            lstrcpy(subBlockName, "\\VarFileInfo\\Translation");
            if (!VerQueryValue(FVData, subBlockName, (void FAR* FAR*)&TransBlock,
&vSize)) {
                delete FVData;
                FVData = 0;
            } else
                // Swap the words so wsprintf will print the lang-charset in the
correct format.
                *(DWORD *)TransBlock = MAKELONG(HIWORD(*(DWORD *)TransBlock),
LOWORD(*(DWORD *)TransBlock));
        }
    }
}


ProjectRCVersion::~ProjectRCVersion ()
{
    if (FVData)
        delete FVData;
}


BOOL ProjectRCVersion::GetProductName (LPSTR &prodName)
{
    UINT    vSize;
    char    subBlockName[255];

    wsprintf(subBlockName, "\\StringFileInfo\\%08lx\\%s", *(DWORD *)TransBlock,
(LPSTR)"ProductName");
    return FVData ? VerQueryValue(FVData, subBlockName, (void FAR* FAR*)&prodName,
&vSize) : FALSE;
}


BOOL ProjectRCVersion::GetProductVersion (LPSTR &prodVersion)
{
    UINT    vSize;
    char    subBlockName[255];

    wsprintf(subBlockName, "\\StringFileInfo\\%08lx\\%s", *(DWORD *)TransBlock,
(LPSTR)"ProductVersion");
        return  FVData  ?  VerQueryValue(FVData,  subBlockName,  (void  FAR*
FAR*)&prodVersion, &vSize) : FALSE;
}


BOOL ProjectRCVersion::GetCopyright (LPSTR &copyright)
{
    UINT    vSize;
    char    subBlockName[255];

    wsprintf(subBlockName, "\\StringFileInfo\\%08lx\\%s", *(DWORD *)TransBlock,
(LPSTR)"LegalCopyright");
        return  FVData  ?  VerQueryValue(FVData,  subBlockName,  (void  FAR*
FAR*)&copyright, &vSize) : FALSE;
}


BOOL ProjectRCVersion::GetDebug (LPSTR &debug)
```

```
{
    UINT    vSize;
    char    subBlockName[255];

     wsprintf(subBlockName, "\\StringFileInfo\\%08lx\\%s", *(DWORD *)TransBlock,
(LPSTR)"SpecialBuild");
    return FVData ? VerQueryValue(FVData, subBlockName, (void FAR* FAR*)&debug,
&vSize) : FALSE;
}


//{{winpwbAboutDlg Implementation}}


//////////////////////////////////////////////////////
// winpwbAboutDlg
// ==========
// Construction/Destruction handling.
static winpwbAboutDlgXfer winpwbAboutDlgData;

winpwbAboutDlg::winpwbAboutDlg (TWindow *parent, TResId resId, TModule *module)
    : TDialog(parent, resId, module)
{
//{{winpwbAboutDlgXFER_USE}}
    autorCtrl = new TStatic(this, IDC_AUTOR, 255);

    SetTransferBuffer(&winpwbAboutDlgData);
//{{winpwbAboutDlgXFER_USE_END}}

    // INSERT>> Your constructor code here.
}


winpwbAboutDlg::~winpwbAboutDlg ()
{
    Destroy();

    // INSERT>> Your destructor code here.
}


void winpwbAboutDlg::SetupWindow ()
{
    LPSTR prodName = 0, prodVersion = 0, copyright = 0, debug = 0;

    // Get the static text for the value based on VERSIONINFO.
    TStatic *versionCtrl = new TStatic(this, IDC_VERSION, 255);
    TStatic *copyrightCtrl = new TStatic(this, IDC_COPYRIGHT, 255);
    TStatic *debugCtrl = new TStatic(this, IDC_DEBUG, 255);

    TDialog::SetupWindow();

    // Process the VERSIONINFO.
    ProjectRCVersion applVersion(GetModule());

    // Get the product name and product version strings.
                        if      (applVersion.GetProductName(prodName)       &&
applVersion.GetProductVersion(prodVersion)) {
        // IDC_VERSION is the product name and version number, the initial value
of IDC_VERSION is
        // the word Version (in whatever language) product name VERSION product
version.
        char    buffer[255];
        char    versionName[128];

        buffer[0] = '\0';
        versionName[0] = '\0';

        versionCtrl->GetText(versionName, sizeof(versionName));
        wsprintf(buffer, "%s %s %s", prodName, versionName, prodVersion);

        versionCtrl->SetText(buffer);
        autorCtrl->SetText("M.Van Asten FUNDP 1995");
    }
```

```
    //Get the legal copyright string.
    if (applVersion.GetCopyright(copyright))
        copyrightCtrl->SetText(copyright);

    // Only get the SpecialBuild text if the VERSIONINFO resource is there.
    if (applVersion.GetDebug(debug))
        debugCtrl->SetText(debug);
}
```

1.1.12 WNPWBAPP.H

```
#if !defined(__wnpwbapp_h)                  // Sentry, use file only if it's not
already included.
#define __wnpwbapp_h

/*  Project winpwb

    Copyright _ 1994. All Rights Reserved.

    SUBSYSTEM:    winpwb.exe Application
    FILE:         wnpwbapp.h
    AUTHOR:       MICHEL VAN ASTEN FUNDP 1995


    OVERVIEW
    ========
    Class definition for winpwbApp (TApplication).
*/


#include <owl\owlpch.h>
#pragma hdrstop

#include "tslavewn.h"
#include <owl\window.h>
#include <owl\statusba.h>
#include <owl\editfile.h>
#include <owl\opensave.h>

#include "wnpwbapp.rh"            // Definition of all resources.
#include "tslave.h"
#include "mode.h"

//
// FrameWindow must be derived to override Paint for Preview and Print.
//
//{{TDecoratedFrame = SDIDecFrame}}
class SDIDecFrame : public TDecoratedFrame {
public:
    SDIDecFrame (  TWindow *parent, const char far *title, TWindow *clientWnd,
BOOL               trackMenuSelection = FALSE, TModule *module = 0);
    ~SDIDecFrame ();

//{{SDIDecFrameVIRTUAL_BEGIN}}
public:
    virtual void Paint (TDC& dc, BOOL erase, TRect& rect);
//{{SDIDecFrameVIRTUAL_END}}
      TSlave       *Slave;
      TStatusBar  *statBar;

//{{SDIDecFrameRSP_TBL_BEGIN}}
protected:
    LRESULT ReadSocket(WPARAM id, LPARAM hWndCtl);
//{{SDIDecFrameRSP_TBL_END}}
DECLARE_RESPONSE_TABLE(SDIDecFrame);
};    //{{SDIDecFrame}}


//{{TApplication = winpwbApp}}
class winpwbApp : public TApplication {
private:
     TOpenSaveDialog::TData FileData;                    // Data to control
open/saveas standard dialog.
```

```
public:
    winpwbApp ();
    virtual ~winpwbApp ();

    void OpenFile (const char *fileName = 0);
// variables
      TSlave *appSlave;


//{{winpwbAppVIRTUAL_BEGIN}}
public:
    virtual void InitMainWindow();
//{{winpwbAppVIRTUAL_END}}

//{{winpwbAppRSP_TBL_BEGIN}}
protected:
    void CmHelpAbout ();
    void doCMmaster ();
    void doCMslave ();

    void doCMAbout ();
//{{winpwbAppRSP_TBL_END}}
  DECLARE_RESPONSE_TABLE(winpwbApp);
};    //{{winpwbApp}}


#endif                                          // __wnpwbapp_h sentry.
```

1.1.13 WNPWBAPP.CPP

```
/*  Project winpwb

    Copyright © 1994. All Rights Reserved.

    SUBSYSTEM:    winpwb.exe Application
    FILE:         wnpwbapp.cpp
    AUTHOR:


    OVERVIEW
    ========
    Source file for implementation of winpwbApp (TApplication).
*/


#include <owl\owlpch.h>
#pragma hdrstop


#include "wnpwbapp.h"
#include "wnpwbabd.h"                           // Definition of about dialog.
#include "distrib.h"


//{{winpwbApp Implementation}}


//
// Build a response table for all messages/commands handled
// by the application.
//
DEFINE_RESPONSE_TABLE1(winpwbApp, TApplication)
//{{winpwbAppRSP_TBL_BEGIN}}
    EV_COMMAND(CM_MASTER, doCMmaster),
       EV_COMMAND(CM_SLAVE, doCMslave),
    EV_COMMAND(CM_ABOUT, doCMAbout),
//{{winpwbAppRSP_TBL_END}}
END_RESPONSE_TABLE;
```

```
//////////////////////////////////////////////////////////
// winpwbApp
// =====
//
winpwbApp::winpwbApp () : TApplication("winWhiteboard")
{
        // INSERT>> Your constructor code here.
        appSlave = NULL;
        // Necessary to use Borland style dialog
        EnableBWCC();
}


winpwbApp::~winpwbApp ()
{
        // INSERT>> Your destructor code here.

}


//////////////////////////////////////////////////////////
// winpwbApp
// =====
// Application intialization.
//
void winpwbApp::InitMainWindow ()
{
        SDIDecFrame *frame = new SDIDecFrame(0, "Win Whiteboard", 0, TRUE); /*
DEBUG */

        nCmdShow = nCmdShow != SW_SHOWMINIMIZED ? SW_SHOWNORMAL : nCmdShow;

        //
        // Assign ICON w/ this application.
        //
        frame->SetIcon(this, IDI_SDIAPPLICATION);

        //
        // Menu associated with window and accelerator table associated with
table.
        //
        frame->AssignMenu(SDI_MENU);

        //
        // Associate with the accelerator table.
        //
        frame->Attr.AccelTable = SDI_MENU;


        TStatusBar *sb = new TStatusBar(      frame, TGadget::Recessed,
                                        TStatusBar::CapsLock        |
                                        TStatusBar::NumLock         |
                                        TStatusBar::ScrollLock      |
                                        TStatusBar::Overtype);
        frame->Insert(*sb, TDecoratedFrame::Bottom);
        frame->statBar = sb;

        SetMainWindow(frame);

}



//
// Build a response table for all messages/commands handled
// by the application.
//
DEFINE_RESPONSE_TABLE1(SDIDecFrame, TDecoratedFrame)
//{{SDIDecFrameRSP_TBL_BEGIN}}
    EV_MESSAGE(WM_SOCKET, ReadSocket),
//{{SDIDecFrameRSP_TBL_END}}
END_RESPONSE_TABLE;
```

```
//{{SDIDecFrame Implementation}}


SDIDecFrame::SDIDecFrame (      TWindow *parent, const char far *title, TWindow
*clientWnd,                     BOOL trackMenuSelection, TModule *module)
                        : TDecoratedFrame( parent, title, clientWnd == 0 ? new
                                           TSlaveWin(0, "") : clientWnd,
                                           trackMenuSelection, module)
{
    // INSERT>> Your constructor code here.

}


SDIDecFrame::~SDIDecFrame ()
{
    // INSERT>> Your destructor code here.

}


////////////////////////////////////////////////////
// winpwbApp
// ===========
// Menu Help About winpwb.exe command
void winpwbApp::CmHelpAbout ()
{
    //
    // Show the modal dialog.
    //
    winpwbAboutDlg(MainWindow).Execute();
}


int OwlMain (int , char* [])
{
    winpwbApp     App;
       int            result;

    result = App.Run();

    return result;
}


void winpwbApp::doCMmaster ()
{
      // INSERT>> Your code here.
      //Mode *pMode;
      //char *address, *slides;

      //Mode(address, slides, hWindow)->Execute();
}


void winpwbApp::doCMslave ()
{
      // INSERT>> Your code here.
      char              address[100], port[100], slides[100];
      TFrameWindow      *notifyWindow;
      TWindow           *slideWindow;
      char              buff[500];

      // Get a pointer on the clientWindow of theMainWindow
      notifyWindow = GetMainWindow();
      slideWindow = GetMainWindow()->GetClientWindow();

      // Check for the presence of an existinsing slave
      if(appSlave){
            LoadString(IDS_EXISTSLAVE, buff, 500);
            if(notifyWindow->MessageBox( buff, "Existing slave", MB_YESNO) ==
IDNO)
                  return;
      }
```

```
        if(Mode( address, port, slides, notifyWindow ).Execute()== IDOK){
                // delete the current slave
                delete appSlave;
                appSlave = new TSlave(notifyWindow, slideWindow, address, port,
slides);
                if (appSlave->GetStatus()== DISTRIBUTION_ERROR){
                        // it's no possible to use the distribution object of the
slave
                        // the slave will be destroy
                        // most important messages are defines in ressource file so
they
                        // will be easy to change if necessary
                        LoadString(IDS_NETERROR, buff, 500);
                        notifyWindow->MessageBox( buff, "Connection error" );
                        delete(appSlave);
                        appSlave = NULL;
        }
    }
}


// when the application receives an incomming packet
// the main window receives a WM_SOCKET message and
// call the slave wbIn function()
//

LRESULT SDIDecFrame::ReadSocket(WPARAM id, LPARAM hWndCtl)
{
        ((winpwbApp*)GetApplication())->appSlave->wbIn();
        BringWindowToTop();
        return TRUE;
}


void winpwbApp::doCMAbout ()
{
        // INSERT>> Your code here.
        winpwbAboutDlg(MainWindow).Execute();
}
```

2. WHITEBOARD

2.1 DITHER.C

```
/* dither.c
 *
 * completely reworked dithering module for xloadimage
 * uses error-diffusion dithering (floyd-steinberg) instead
 * of simple 4x4 ordered-dither that was previously used
 *
 * the previous version of this code was written by steve losen
 * (scl@virginia.edu)
 *
 * jim frost     07.10.89
 * Steve Losen   11.17.89
 * kirk johnson 06.04.90
 *
 * Copyright 1990 Kirk L. Johnson (see the included file
 * "kljcpyrght.h" for complete copyright information)
 *
 * Copyright 1989, 1990 Jim Frost and Steve Losen.  See included file
 * "copyright.h" for complete copyright information.
 */

/* MODIF_MVA include RGBMap.h not image.h */
#include "RGBMap.h"
```

```
#define RedPercent  0.299
#define GrnPercent  0.587 /* color -> grey conversion parameters */
#define BluPercent  0.114

#define MaxIntensity  65536     /* maximum possible Intensity */

#define MaxGrey       32768     /* limits on the grey levels used */
#define Threshold     16384     /* in the dithering process */
#define MinGrey           0

static unsigned int tone_scale_adjust();
static void         LeftToRight();
static void         RightToLeft();


/*
 * simple floyd-steinberg dither with serpentine raster processing
 */

Image *dither(cimage, verbose)
     Image        *cimage;
     unsigned int  verbose;
{
      Image          *image;    /* destination image */
      unsigned int   *grey;     /* grey map for source image */
      double         tmp;       /* work space */
      unsigned int   spl;       /* source pixel length in bytes */
      unsigned int   dll;       /* destination line length in bytes */
      unsigned char  *src;      /* source data */
      unsigned char  *dst;      /* destination data */
      int            *curr;     /* current line buffer */
      int            *next;     /* next line buffer */
      int            *swap;     /* for swapping line buffers */
      Pixel          color;     /* pixel color */
      unsigned int   level;     /* grey level */
      unsigned int   i, j;      /* loop counters */

  /*
   * check the source image
   */

/*
      goodImage(cimage, "dither");
      if (! RGBP(cimage))
            return(NULL);
*/

  /*
   * allocate destination image
   */


/*
      if (verbose)
      {
            printf("  Dithering image...");
            fflush(stdout);
      }

*/
      image = newBitImage(cimage->width, cimage->height);
      if (cimage->title)
      {
            image->title = (char *)lmalloc(strlen(cimage->title) + 12);
            sprintf(image->title, "%s (dithered)", cimage->title);
      }

  /*
   * if the number of entries in the colormap isn't too large, compute
   * the grey level for each entry and store it in grey[]. else the
   * grey levels will be computed on the fly.
   */

/* BEGIN_MODIF_MVA */
/* no dithering needed */
```

```
        if (cimage->depth > 1)
        {
                if (cimage->depth <= 16)
                {
                        grey = (unsigned int *)lmalloc(sizeof(unsigned int)
                                * cimage->rgb.used);
                        for (i=0; i<cimage->rgb.used; i++)
                        {
                                tmp  = (cimage->rgb.red[i]   * RedPercent);
                                tmp += (cimage->rgb.green[i] * GrnPercent);
                                tmp += (cimage->rgb.blue[i]  * BluPercent);

                                grey[i] = (tmp / MaxIntensity) * MaxGrey;
                        }

                        for (i=0; i<cimage->rgb.used; i++)
                                grey[i] = tone_scale_adjust(grey[i]);
                }
                else
                {
                        grey = NULL;
                }
        }
/* END_MODIF_MVA */

  /*
   * dither setup
   */
        spl = cimage->pixlen;
        dll = (image->width / 8) + (image->width % 8 ? 1 : 0);
        src = cimage->data;
        dst = image->data;

/* BEGIN_MODIF_MVA */
        if (cimage->depth > 1)
        {
                curr  = (int *)lmalloc(sizeof(int) * (cimage->width + 2));
                next  = (int *)lmalloc(sizeof(int) * (cimage->width + 2));
                curr += 1;
                next += 1;
                for (j=0; j<cimage->width; j++)
                {
                        curr[j] = 0;
                        next[j] = 0;
                }
        }

        /* depth = 1 no dithering needed it's only necessary*/

        if (cimage->depth == 1)
        {
                for (i=0; i<cimage->height; i++)
                {
                /* reduce the current line */
                /* copy the dithered line to the destination image */
                        for (j=0; j<cimage->width; j++)
                                if(src[j] & 1){
                                        /* initial solution ...*/
                                        /* dst[j / 8] |= 1 << (7 - (j & 7)); */
                                        dst[j / 8] |= 1 << (j & 7); /* modification */
                                }
                        dst += dll;
                        src += cimage->width;
                }

  /*
   * clean up
   */

                freeImage(cimage);
                return(image);
        }

  /* END_MODIF_MVA */
```

```
  /*
   * primary dither loop
   */
      for (i=0; i<cimage->height; i++)
      {
            /* copy the row into the current line */
            for (j=0; j<cimage->width; j++)
            {
                  color = memToVal(src, spl);
                  src  += spl;

                  if (grey == NULL)
                  {
                        tmp  = (cimage->rgb.red[color]   * RedPercent);
                        tmp += (cimage->rgb.green[color] * GrnPercent);
                        tmp += (cimage->rgb.blue[color]  * BluPercent);

                        level   =   tone_scale_adjust((tmp   /   MaxIntensity)   *
MaxGrey);
                  }
                  else
                  {
                        level = grey[color];
                  }

                  curr[j] += level;
            }

            /* dither the current line */
            if (i & 0x01)
                  RightToLeft(curr, next, cimage->width);
            else
                  LeftToRight(curr, next, cimage->width);

            /* copy the dithered line to the destination image */
            for (j=0; j<cimage->width; j++)
                  if (curr[j] > Threshold) /* avant < */
                        /* BEGIN_MODIF_MVA */
                        /* initial solution */
                        /* dst[j / 8] |= 1 << (7 - (j & 7)); */
                        dst[j / 8] |= 1 << (j & 7); /* new solution*/
                        /* END_MODIF_MVA */
                  dst += dll;

            /* circulate the line buffers */
            swap = curr;
            curr = next;
            next = swap;
            for (j=0; j<cimage->width; j++)
                  next[j] = 0;
      }

      /*
      * clean up
      */
      lfree(grey);
      lfree(curr-1);
      lfree(next-1);
      /* BEGIN_MODIF_MVA */
      /* free the initial image, no more needed */
      freeImage(cimage);

      /* END_MODIF_MVA */
      return(image);
}


/*
 * a _very_ simple tone scale adjustment routine. provides a piecewise
 * linear mapping according to the following:
 *
 *      input:          output:
```

```
 *      0 (MinGrey)     0 (MinGrey)
 *      Threshold       Threshold/2
 *      MaxGrey         MaxGrey
 *
 * this should help things look a bit better on most displays.
 */
static unsigned int tone_scale_adjust(val)
     unsigned int val;
{
      unsigned int rslt;

      if (val < Threshold)
            rslt = val / 2;
      else
            rslt = (((val - Threshold) * (MaxGrey-(Threshold/2))) /
            (MaxGrey-Threshold)) + (Threshold/2);

      return rslt;
}


/*
 * dither a line from left to right
 */
static void LeftToRight(curr, next, width)
     int *curr;
     int *next;
     int  width;
{
      int idx;
      int error;
      int output;

      for (idx=0; idx<width; idx++)
      {
            output       = (curr[idx] > Threshold) ? MaxGrey : MinGrey;
            error        = curr[idx] - output;
            curr[idx]    = output;
            next[idx-1] += error * 3 / 16;
            next[idx]   += error * 5 / 16;
            next[idx+1] += error * 1 / 16;
            curr[idx+1] += error * 7 / 16;
      }
}


/*
 * dither a line from right to left
 */
static void RightToLeft(curr, next, width)
     int *curr;
     int *next;
     int  width;
{
      int idx;
      int error;
      int output;

      for (idx=(width-1); idx>=0; idx--)
      {
            output       = (curr[idx] > Threshold) ? MaxGrey : MinGrey;
            error        = curr[idx] - output;
            curr[idx]    = output;
            next[idx+1] += error * 3 / 16;
            next[idx]   += error * 5 / 16;
            next[idx-1] += error * 1 / 16;
            curr[idx-1] += error * 7 / 16;
      }
}


/*    MODIF_MVA : the next function where not included in the initial version
      but they are necessary for the comprehension of dither function */
unsigned long memToVal(p, len)
     byte        *p;
```

```
    unsigned int  len;
{
    unsigned int  a;
    unsigned long i;

    i= 0;
    for (a= 0; a < len; a++)
        i= (i << 8) + *(p++);
    return(i);
}

void valToMem(val, p, len)
    unsigned long  val;
    byte           *p;
    unsigned int   len;
{
    int a;

    for (a= len - 1; a >= 0; a--) {
        *(p + a)= val & 0xff;
        val >>= 8;
    }
}

Image *newRGBImage(width, height, depth)
    unsigned int width, height, depth;
{
    Image        *image;
    unsigned int  pixlen, numcolors, a;

    pixlen= (depth / 8) + (depth % 8 ? 1 : 0);
    for (numcolors= 2, a= depth - 1; a; a--)
        numcolors *= 2;
    image= (Image *)lmalloc(sizeof(Image));
    image->type= IRGB;
    image->title= NULL;
    newRGBMapData(&(image->rgb), numcolors);
    image->width= width;
    image->height= height;
    image->depth= depth;
    image->pixlen= pixlen;
    image->data= (unsigned char *)lmalloc(width * height * pixlen);
    return(image);
}


void newRGBMapData(rgb, size)
    RGBMap       *rgb;
    unsigned int  size;
{
    rgb->used= 0;
    rgb->size= size;
    rgb->red= (Intensity *)lmalloc(sizeof(Intensity) * size);
    rgb->green= (Intensity *)lmalloc(sizeof(Intensity) * size);
    rgb->blue= (Intensity *)lmalloc(sizeof(Intensity) * size);
}

void freeRGBMapData(rgb)
    RGBMap *rgb;
{
    lfree(rgb->red);
    lfree(rgb->green);
    lfree(rgb->blue);
}

Image *newBitImage(width, height)
    unsigned int width, height;
{
    Image        *image;
    unsigned int  linelen;

    image= (Image *)lmalloc(sizeof(Image));
    image->type= IBITMAP;
    image->title= NULL;
    newRGBMapData(&(image->rgb), (unsigned int)2);
```

```
        *(image->rgb.red)= *(image->rgb.green)= *(image->rgb.blue)= 65535;
        *(image->rgb.red + 1)= *(image->rgb.green + 1)= *(image->rgb.blue + 1)= 0;
        image->rgb.used= 2;
        image->width= width;
        image->height= height;
        image->depth= 1;
        linelen= (width / 8) + (width % 8 ? 1 : 0); /* thanx johnh@amcc.com */
        image->data= (unsigned char *)lcalloc(linelen * height);
        return(image);
}

void freeImageData(image)
     Image *image;
{
        if (image->title) {
                lfree(image->title);
                image->title= NULL;
        }
        freeRGBMapData(&(image->rgb));
        lfree(image->data);
}

void freeImage(image)
     Image *image;
{
        freeImageData(image);
        lfree(image);
}

byte *lmalloc(size)
     unsigned int size;
{
        byte *area;

        if (!(area= (byte *)malloc(size))) {
                perror("malloc");
                exit(1);
        }
        return(area);
}

byte *lcalloc(size)
     unsigned int size;
{
        byte *area;

        if (!(area= (byte *)calloc(1, size))) {
                perror("calloc");
                exit(1);
        }
        return(area);
}

void lfree(area)
     byte *area;
{
        free(area);
}
```

2.2 GIFREADM.C

la définition de la fonction setImage est la seule modification apportée au code initial, la fonction est appelée à la fin de l'exécution de la fonction loadGif qui retourne maintenant un pointeur sur une structure de type Image.

```
     /* BEGIN_MODIF_MVA */

     Image *setImage(Image *pImage)
     {
```

```
    pImage   = (Image*) malloc (sizeof(Image));

    pImage->title = NULL;
    pImage->rgb.size = im_width * im_height * im_pixlen;
    pImage->rgb.red = im_cmap->red;
    pImage->rgb.blue = im_cmap->blue;
    pImage->rgb.green = im_cmap->green;
    pImage->width = im_width;
    pImage->height = im_height;
    pImage->depth = im_depth;
    pImage->pixlen = im_pixlen;
    pImage->data = im_data;
    pImage->type = IRGB;
    pImage->rgb.used = im_cmap->used;

    printf("im_cmap->used %d\n",im_cmap->used);

    printf("End functiom SetImage\n");
    return pImage;
  }

/* END_MODIF_MVA */
```